24th European Conference on Artificial Intelligence - ECAI 2020
Santiago de Compostela, Spain

Agent Abstraction via Forgetting in the Situation Calculus

Kailun Luo[1] and **Yongmei Liu**[*1] and **Yves Lespérance**[2] and **Ziliang Lin**[1]

Abstract. In an earlier paper, Banihashemi et al. proposed a general framework for agent abstraction based on the situation calculus. They used basic action theories (BATs) to represent agents' behavior, and mappings to specify how high-level BATs relate to low-level ones. They then defined the concepts of sound/complete abstractions of BATs based on the notion of bisimulation between high-level and low-level models. However, they didn't address the issue of the construction of an abstraction from a low-level action theory when given a mapping. It turns out that their concept of abstraction is closely related to the well-explored notion of forgetting. In this paper, we explore agent abstraction via forgetting. Firstly, we show that a correct (i.e., sound and complete) abstraction can be obtained via forgetting low-level symbols from the low-level action theory together with axioms for bisimulation. Secondly, we show how to compute via forgetting a correct abstraction in the form of a generalized BAT (i.e., where the initial database, action preconditions and successor state descriptions can be second-order formulas) under a suitable Markovian restriction. Finally, we show that in the propositional case, under the suitable Markovian restriction, correct abstractions are always computable.

1 Introduction

Abstraction plays an important role in problem solving. Abstraction is used as a heuristic to guide the searching process in AI planning [11, 10], as a technique to deal with computational complexity in program verification [7, 18] and model checking [4, 3], and as a method to form a general solution in generalized planning [21, 2].

Recently, Banihashemi *et al.* [1] proposed a general *agent abstraction* framework based on the situation calculus [16, 19] and the ConGolog agent programming language [5]. In this framework, they assumed that there is a high-level basic action theory (BAT), a low-level BAT, and a refinement mapping. The two theories represent an agent's behavior at different levels of detail. The refinement mapping relates the two theories by associating high-level symbols to more detailed low-level representations, such that low-level programs are abstracted as actions and low-level properties (represented by formulas) are abstracted as fluents. Moreover, based on the concept of bisimulation [17] between high-level and low-level models, they defined concepts of sound/complete abstractions of low-level BATs under given refinement mappings.

Agent abstraction provides a framework in which one can formalize a *good* high-level account of an agent's possible low-level behavior. At the high level, one can suppress details and build complex concepts from the low level. This can be used to plan more efficiently, to explain the agent's behavior in high-level terms, etc. However, it is not obvious how to verify that there is a sound and/or complete abstraction for a low-level BAT given a refinement mapping, and how to construct one. To illustrate this, consider mobile robots where high-level primitive actions contain loops and branches that build on low-level motor commands. If a designer provides a candidate mapping indicating the relation between the high-level actions and their low-level representations, then one would want to know whether this yields a correct (i.e., sound and complete) abstraction.

In this paper, we fill the gap by relating agent abstraction to the notion of forgetting in first-order/second-order logics [14]. Similar to abstraction that removes unimportant details, forgetting a symbol aims at omitting the information of the symbol while retaining all the facts that are *irrelevant* to the symbol. Lin and Reiter [14] proposed forgetting to update the database in response to a performed action. Later, many definitions of forgetting operators were introduced in other logics, *e.g.*, modal logics and [24] logic programming [23, 9]. Forgetting has been studied extensively and techniques have been developed to express and compute it [6].

We first investigate necessary and sufficient conditions under which there exists a correct abstraction, given a low-level BAT and a refinement mapping. These conditions are needed because we focus on situation calculus models and Markovian theories. At the model level, we identify a condition for the existence of a *correct* high-level model. Roughly, the restriction says that when abstracting a (possibly non-deterministic) low-level program to a high-level action, the different executions of the program should be *indistinguishable* in the high-level language, since atomic actions in the situation calculus are assumed to be deterministic. At the theory level, to guarantee that high-level models are uniformly characterized in the same BAT which has the *Markov property* (i.e., the executability conditions and the effects of actions are fully determined by the present state of the system) [8], we identify a suitable Markovian restriction at the low-level. The restriction says that, at the low-level, for the programs that represent high-level actions, the executability conditions and effects/postconditions should only be determined by the *abstract states* formed by the low-level formulas that represent high-level fluents.

We then show how to construct an abstraction via forgetting. Firstly, we show that a correct abstraction can be obtained via forgetting low-level symbols from the low-level BAT together with axioms for bisimulation [17]. Secondly, we show how to use forgetting to compute a correct abstraction in the form of a generalized BAT (i.e., where the initial database and the action preconditions and successor state axioms can be second-order formulas) under the Markovian

[1] Dept. of Computer Science, Sun Yat-sen University, Guangzhou, China, email: luokl3@mail2.sysu.edu.cn & ymliu@mail.sysu.edu.cn & linzlang@mail2.sysu.edu.cn, *Corresponding author
[2] Dept. of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada, email:lesperan@cse.yorku.ca

restriction. Finally, we show that in the propositional case, under the Markovian restriction, correct abstractions are always computable.

2 Preliminaries

2.1 Situation Calculus and Basic Action Theories

The $situation\ calculus\ \mathcal{L}$ is a first-order (FO) language with limited second-order features for representing dynamically changing worlds [16, 19]. In a situation calculus language, there are three disjoint sorts: $action$ for actions, $situation$ for situations and $object$ for everything else. There are some special symbols: situation constant S_0 denotes the initial situation; binary function $do(a, s)$ denotes the situation resulting from performing action a in situation s; relation $Poss(a, s)$ states that it is possible to perform action a in situation s. Dynamic properties are captured by predicates called $fluents$ whose truth values vary from situation to situation (we omit functional fluents here). A $situation$-$suppressed$ formula is one where all the situation arguments of fluents are dropped and thus no situation is mentioned. If φ is a situation-suppressed formula, then we use $\varphi[s]$ to denote the formula obtained from φ by restoring s as the situation arguments to all fluents. Often, we consider a *uniform* formula which refers to a particular situation τ.

Within the language, a particular domain can be specified by a basic action theory (BAT) of the form:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}, \text{where}$$

- Σ is the set of the foundational axioms for situations;
- $\mathcal{D}_{una}$ is the set of unique name axioms for actions;
- $\mathcal{D}_{ap}$ is a set of precondition axioms of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$, where $\Pi_A(\vec{x}, s)$ is a FO uniform formula, stating when action $A(\vec{x})$ can be legally performed;
- $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs) of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a FO uniform formula, describing how fluents are changed by actions;
- $\mathcal{D}_{S_0}$ is a set of FO sentences as the initial database, where the only situation term that appears is S_0.

We use the blocks world as our running example. The blocks world consists of a finite number of blocks stacked into towers on a table [20]. A blocks world problem is to turn an initial state of the blocks into a goal state, by moving one block at a time from the top of a tower or from the table onto another tower or to the table.

We formalize the blocks world as a BAT, where precondition axioms and SSAs are as follows :

Example 1 In the blocks world, action $stack(x, y)$ means moving block x onto block y; action $movetotable(x)$ means moving block x to the table. Fluent $clear(x, s)$ means that block x has no other blocks on top of it, in situation s; fluent $on(x, y, s)$ means that block x is on block y, in situation s; fluent $ontable(x, s)$ means that block x is on the table, in situation s.

$$Poss(movetotable(x), s) \equiv \neg ontable(x, s) \wedge clear(x, s);$$
$$Poss(stack(x, y), s) \equiv x \neq y \wedge clear(x, s) \wedge clear(y, s);$$
$$clear(x, do(a, s)) \equiv \{\exists y. on(y, x, s) \wedge [\exists z. a = stack(y, z)$$
$$\vee\ a = movetotable(y)] \vee clear(x, s) \wedge \forall y. a \neq stack(y, x)\};$$
$$ontable(x, do(a, s)) \equiv \{a = movetotable(x)$$
$$\vee\ ontable(x, s) \wedge \forall y. a \neq stack(x, y)\};$$
$$on(x, y, do(a, s)) \equiv \{a = stack(x, y) \vee on(x, y, s)$$
$$\wedge\ a \neq movetotable(x) \wedge [\forall z. a = stack(x, z) \supset z = y]\}.$$

2.2 High-level Programming Language Golog

To represent and reason about complex actions, Levesque *et al.* [12] introduced a high-level programming language called Golog. The syntax of Golog is as follows:

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

where α is an action term, possibly with parameters; formula φ is a situation-suppressed formula and $\varphi?$ tests whether φ holds; program $\delta_1; \delta_2$ represents the sequential execution of δ_1 and δ_2; program $\delta_1 | \delta_2$ denotes the non-deterministic choice between δ_1 and δ_2; program $\pi x.\delta$ denotes the non-deterministic choice of a value for parameter x in δ; program δ^* means executing program δ zero or more times.

The semantics of Golog is specified by $Do(\delta, s, s')$, which macro-expands into a situation calculus formula, saying that it is possible to reach situation s' from situation s by executing a sequence of actions specified by program δ.

2.3 Regression and Progression via Forgetting

Regression and progression are two important mechanisms for reasoning about actions and their effects. Regression was proposed by Reiter [19] to reduce the evaluation of a sentence to a FO theorem-proving task in the initial database.

Here we present one-step regression for $regressable$ formulas. The essence of a regressable formula is that each of its situation terms is rooted at a certain situation s, and therefore, one can tell, by inspection of such a term, exactly how many actions it involves [19].

Definition 1 We use $\mathcal{R}_{\mathcal{D}}[\phi]$ to denote the formula obtained from regressable formula ϕ by replacing each fluent atom $F(\vec{t}, do(\alpha, \sigma))$ with $\Phi_F(\vec{t}, \alpha, \sigma)$ and each precondition atom $Poss(A(\vec{t}), \sigma)$ with $\Pi_A(\vec{t}, \sigma)$, and further simplifying the result with $\mathcal{D}_{una}$.

Proposition 1 $\mathcal{D} \models \phi \equiv \mathcal{R}_{\mathcal{D}}[\phi]$.

Instead, progression means updating an agent's knowledge base after an action has been performed. Lin and Reiter [15] formalized the notion of progression. Let α be a ground action, and let S_α denote the situation term $do(\alpha, S_0)$.

Definition 2 Let M and M' be two structures (for the situation calculus language) with the same domains for sorts $action$ and $object$. We say that M and M' are isomorphic at S_α, written $M' \sim_{S_\alpha} M$ iff the following two conditions hold:

1. M' and M interpret all situation-independent predicate and function symbols identically;
2. M' and M agree on all fluents at $\mathcal{S}_\alpha$: for every fluent F, and every variable assignment v,
$$M', v \models F(\vec{x}, do(\alpha, S_0)) \text{ iff } M, v \models F(\vec{x}, do(\alpha, S_0)).$$

Let $\mathcal{L}_{S_\alpha}$ be the subset of $\mathcal{L}$ that mentions no other situation terms except S_α. Let $\mathcal{L}^2_{S_\alpha}$ be the second-order extension of $\mathcal{L}_{S_\alpha}$.

Definition 3 A set of sentences $\mathcal{D}_{S_\alpha}$ in $\mathcal{L}^2_{S_\alpha}$ is a progression of the initial database $\mathcal{D}_{S_0}$ to S_α (with respect to $\mathcal{D}$) iff for any structure M, M is a model of $\mathcal{D}_{S_\alpha}$ iff there is a model M' of $\mathcal{D}$ such that $M \sim_{S_\alpha} M'$.

Progression can be viewed as a result of forgetting. The agent needs to forget about all the facts that are no longer true, and in such a way that this will not affect any of her reasoning about possible future actions [14]. This relies on the following semantic definition of forgetting. Note that forgetting about a predicate is a second-order notion. All theories and formulas are assumed to be in the second-order language.

Definition 4 Let μ be a ground atom or a predicate. Let M_1 and M_2 be two structures (for the second-order language). We say M_1, M_2 are *the same* except for μ, written $M_1 \sim_\mu M_2$, if M_1 and M_2 agree on everything except possibly on the interpretation of μ.

Definition 5 Let T be a theory and μ be a ground atom or a predicate. A theory T' is a result of forgetting about μ in T, denoted by $forget(T;\mu)$, if for any structure M, $M \models T'$ iff there is a model M' of T such that $M \sim_\mu M'$.

Forgetting a symbol results in a weaker theory which entails the same set of sentences that are $irrelevant$ to the symbol.

Proposition 2 *Let T be a theory and μ be a ground atom or a predicate. For any formula φ that does not mention μ, $T \models \varphi$ iff $forget(T;\mu) \models \varphi$;*

Example 2 A result of forgetting about atom $ontable(B)$ from theory $T \doteq \{\forall x.ontable(x)\}$ is $\{\forall x.x \neq B \supset ontable(x)\}$. If we forget about predicate $ontable$ from T, then a result is $\{\exists R \forall x.R(x)\}$.

Lin and Reiter [15] showed that progression is not always first-order definable but is always second-order definable. Here we present a variant of the progression theorem.

Theorem 1 *A progression of a formula ϕ wrt action α and situation σ related to theory $\mathcal{D}$, denoted as $\mathcal{P}_\mathcal{D}(\phi[\sigma], \alpha)$, is the following:*

$$\exists \vec{R}.\{\phi[\sigma] \wedge \bigwedge(\mathcal{D}_{una} \cup \mathcal{D}_{ss}[\alpha, \sigma]) \uparrow \sigma\},$$

where $\varphi \uparrow \sigma$ means replacing every fluent of the form $F(\vec{t}, \sigma)$ in φ by a new predicate variable $R(\vec{t})$, and $\mathcal{D}_{ss}[\alpha, \sigma]$ is the instantiation of $\mathcal{D}_{ss}$ on α and σ.

The above progression result can be viewed as a result of forgetting all the predicates mentioning situation σ, from formula $\phi[\sigma]$ together with the instantiated SSAs.

2.4 Agent Abstraction Framework

In the agent abstraction framework, Banihashemi *et al.* [1] assumed that there is a high-level action theory $\mathcal{D}_h$, a low-level action theory $\mathcal{D}_l$ and a refinement mapping, where $\mathcal{D}_h$ (resp.$\mathcal{D}_l$) contains a finite set of action functions $\mathcal{A}_h$ (resp.$\mathcal{A}_l$) and a finite set of fluent predicates $\mathcal{F}_h$ (resp.$\mathcal{F}_l$).

In this paper, instead of ConGolog [5], we only consider Golog programs as refinements of high-level actions.

Definition 6 A refinement mapping m is a function that

- maps each situation-suppressed high-level fluent $\mathbf{F}$ in $\mathcal{F}_h$ to a situation-suppressed formula ϕ_F defined over $\mathcal{F}_l$;
- associates each high-level action function $\mathbf{A}$ in $\mathcal{A}_h$ to a Golog program δ_A defined over $\mathcal{A}_l$ and $\mathcal{F}_l$.

Example 3 Given the blocks world formalized as in Example 1 as the low-level theory, at the high-level, we focus on a property that all blocks are on the table, and a program that moves any number of blocks onto the table. We use high-level fluent $allontable$ to denote the property and high-level action $moveany$ to denote the program. Then, a refinement mapping m is given as follows:

$$m(allontable) = \forall x.ontable(x),$$
$$m(moveany) = [(\pi x)movetotable(x)]^*.$$

Given a refinement mapping m, they defined an isomorphism relation called m-$isomorphic$ between a high-level situation and a low-level situation, as follows:

Definition 7 Given a refinement mapping m, a situation s_h of a high-level model M_h and a situation s_l of a low-level model M_l, we say that s_h is m-isomorphic to s_l, written $s_h \sim_m^{M_h, M_l} s_l$, if for any high-level fluent $\mathbf{F}$ in $\mathcal{F}_h$ and any variable assignment v,

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

where $v[s/s']$ denotes an assignment which maps s to s' and is the same as v elsewhere.

Based on m-isomorphism, they defined an m-$bisimulation$ relation between a high-level model and a low-level model. Intuitively, an m-bisimulation relation says that with the refinement mapping, two models behave the same if the low-level model is viewed in an abstract way that takes low-level programs as high-level actions and low-level formulas as high-level fluents.

Let Δ^M denote the domain of situations in M, and notation S_0^M stands for the denotation of S_0 in model M. Formally,

Definition 8 A relation $B \subseteq \Delta^{M_h} \times \Delta^{M_l}$ is an m-bisimulation relation between models M_h and M_l if $\langle s_h, s_l \rangle \in B$ implies:

- Situations s_h and s_l are m-isomorphic;
- For any action $\mathbf{A}$ in $\mathcal{A}_h$, if there exists s_h' such that $M_h, v[s/s_h, s'/s_h'] \models Poss(\mathbf{A}(\vec{x}), s) \wedge s' = do(\mathbf{A}(\vec{x}), s)$, then there exists s_l' such that $M_l, v[s/s_l, s'/s_l'] \models Do(m(\mathbf{A})(\vec{x}), s, s')$ and $\langle s_h', s_l' \rangle \in B$;
- For any action $\mathbf{A}$ in $\mathcal{A}_h$, if there exists s_l' such that $M_l, v[s/s_l, s'/s_l'] \models Do(m(\mathbf{A})(\vec{x}), s, s')$, then there exists s_h' such that $M_h, v[s/s_h, s'/s_h'] \models Poss(\mathbf{A}(\vec{x}), s) \wedge s' = do(\mathbf{A}(\vec{x}), s)$ and $\langle s_h', s_l' \rangle \in B$.

If $(S_0^{M_h}, S_0^{M_l}) \in B$, where B is an m-bisimulation relation between M_h and M_l, then M_h is m-bisimilar to M_l, written $M_h \sim_m M_l$.

At the theory level, they defined the notion of a high-level action theory being a sound/complete abstraction of a low-level action theory. *Sound abstraction* says that every low-level (concrete) model has an m-bisimilar high-level (abstract) model, that is,

$$\forall M_l \models \mathcal{D}_l, \exists M_h \models \mathcal{D}_h \text{ s.t. } M_h \sim_m M_l.$$

Complete abstraction means that every high-level model has an m-bisimilar low-level model, that is,

$$\forall M_h \models \mathcal{D}_h, \exists M_l \models \mathcal{D}_l \text{ s.t. } M_h \sim_m M_l.$$

With sound abstractions, one can reason about actions (e.g., their executability) at the abstract level, and refine and concretely execute them at the low level.

2.5 Least Fixed Point Logic

We present least fixed-point (LFP) logic for our later extension of regression and progression with loops.

Definition 9 If $\varphi(R, \vec{x})$ is a formula with free first-order variables $\vec{x} := x_1, \ldots, x_k$ and a free second-order variable R of arity k such that R occurs only within an even number of nested negations in φ, then $[\mathbf{lfp}_{R,\vec{x}}\varphi](\vec{t})$ is also a formula in LFP logic, where $\vec{t}$ is a tuple of terms of the same length as $\vec{x}$.

Definition 10 Let $\varphi(R, \vec{x})$ be as in Def. 9. Let M be a structure providing an interpretation of the free variables of φ except for $\vec{x}$. Then for any tuple $\vec{t}$ of terms, $M \models [\mathbf{lfp}_{R,\vec{x}}\varphi](\vec{t})$ iff $\vec{t}^M$ belongs to the least fixed point formed by $relational\ operation$ F_φ which transforms relation R to new relation $\{\vec{e} : (M, R) \models \varphi[\vec{e}]\}$.

According to the Knaster and Tarski Theorem [22], the least fixed point can be approximated by constructing an inductive fixed point as follows: $R^0 := \emptyset$ and $R^{\alpha+1} := F_\varphi(R^\alpha)$, which iteratively generates R^i until the least ordinal α such that $R^{\alpha+1} = R^\alpha$.

3 Characterizing Abstractions via Forgetting

In this section, we present a necessary and sufficient condition for using forgetting to characterize the semantics of a correct (i.e., sound and complete) abstraction, and we show how to obtain correct abstractions via forgetting.

3.1 Non-Deterministic Uniform Restriction

Not every refinement mapping is suitable for constructing a correct abstraction, since actions in the situation calculus are assumed to be deterministic while Golog programs may be non-deterministic. This may cause inconsistency when trying to abstract a program to an action. We illustrate this with an example.

Example 3 cont'd Suppose the blocks world $\mathcal{D}_l$ has the following initial database: there are two blocks named B_1 and B_2, where B_1 is on the table and B_2 is on B_1. That is,

$$\mathcal{D}_{S_0} = \{\forall x \forall y. x = B_2 \wedge y = B_1 \equiv on(x, y, S_0), B_1 \neq B_2,\\ \forall x. x = B_1 \equiv ontable(x, S_0),\ \forall x. x = B_2 \equiv clear(x, S_0)\}.$$

We show that there is no correct abstraction for $\mathcal{D}_l$, as there is no m-bisimilar high-level model for a certain low-level model. Consider a low-level model M_l with domain $\{B_1, B_2\}$. To construct M_h, in the initial situation S_0, we know that fluent $allontable$ is $false$ since block B_2 is not on the table. However, in the next situation after performing action $moveany$, the valuation on $allontable$ yields a contradiction. That is, the valuation depends on executing the refinement of action $moveany$ at the low-level, which involves non-deterministic iteration of program $(\pi x)movetotable(x)$: if the number of iterations is zero, then $allontable$ is still $false$; otherwise $allontable$ is $true$.

In the following, closely related to Corollary 5 in [1], we identify a necessary and sufficient condition called *non-deterministic uniformity* (NDU) for a mapping given a low-level action theory. Intuitively, it requires that different refinements of any high-level action sequence are indistinguishable in the high-level language:

Definition 11 Given a refinement mapping m and a low-level action theory $\mathcal{D}_l$, we say m is NDU, if for any ground high-level action sequence $\vec{\alpha}$, action $\mathbf{A}$ in $\mathcal{A}_h$, fluent $\mathbf{F}$ in $\mathcal{F}_h$ and variable assignment v,

$$\mathcal{D}_l, v \models \forall s_1, s_2. Do(m(\vec{\alpha}), S_0, s_1) \wedge Do(m(\vec{\alpha}), S_0, s_2) \supset \\ \{\exists s_1'. Do(m(\mathbf{A})(\vec{x}), s_1, s_1') \equiv \exists s_2'. Do(m(\mathbf{A})(\vec{x}), s_2, s_2')\}; \quad (1)$$

$$\mathcal{D}_l, v \models \forall s_1, s_2. Do(m(\vec{\alpha}), S_0, s_1) \wedge Do(m(\vec{\alpha}), S_0, s_2) \supset \\ \{m(\mathbf{F})(\vec{y}, s_1) \equiv m(\mathbf{F})(\vec{y}, s_2)\}, \quad (2)$$

where $m(\vec{\alpha})$ results from refining every high-level action $\mathbf{A}$ in $\vec{\alpha}$ to the low-level program $m(\mathbf{A})$.

Proposition 3 *For the existence of sound and complete abstractions, NDU for refinement mappings is necessary.*

Proof: It is proved by contradiction. If a mapping is not NDU and suppose that it violates condition (1) in Def .11, then there exist two low-level situations s_1, s_2 that are two different refinements of the same high-level situation s_h, and that make $\exists s_1'. Do(m(\mathbf{A})(\vec{x}), s_1, s_1') \not\equiv \exists s_2'. Do(m(\mathbf{A})(\vec{x}), s_2, s_2')$. With m-bisimulation, we have that $Poss(\mathbf{A}, s_h) \not\equiv Poss(\mathbf{A}, s_h)$, which comes to a contradiction. The case where (2) is violated can be handled similarly. ■

Example 3 cont'd We revise m by adding a test action at the ending of the non-deterministic iteration to make it be NDU:

$$m(moveany) = [(\pi x) movetotable(x)]^*; \forall x. ontable(x)?,$$

since iterating only zero times in the previous case will be eliminated.

3.2 Abstraction via Forgetting under NDU

We show how to characterize the semantics of correct abstractions via forgetting under the NDU restriction.

To apply forgetting in dynamic theories, we introduce two notions: forgetting a fluent and forgetting an action function. Since fluents are special predicates, we reduce forgetting a fluent to the case of forgetting a predicate. The intuition of forgetting a fluent is to forget the interpretation of the fluent in all situations. Forgetting an action function means forgetting the denotation of the action function. In fact, at the model level, forgetting an action function does not eliminate any action, but just changes the denotation of actions. Compared to predicates, we introduce forgetting a function by modifying Def. 4 and Def. 5 to allow μ to be an action function.

Example 4 A result of forgetting the action function $move$ from the theory $\{\exists x \exists y. ontable(x, do(move(x, y), S_0))\}$ is the theory $\{\exists f \exists x \exists y. ontable(x, do(f(x, y), S_0))\}$.

To characterize sound abstraction and complete abstraction, we divide forgetting into sound forgetting and complete forgetting.

Definition 12 Let T be a theory and μ be a relation or a function. A theory T' is a result of sound forgetting about μ in T, written $T' \in sforget(T; \mu)$, if for any structure M, if there is a model M' of T s.t. $M' \sim_\mu M$, then $M \models T'$.

Definition 13 Let T be a theory and μ be a relation or function. A theory T' is a result of complete forgetting about μ in T, written $T' \in cforget(T; \mu)$, if for any structure M, if $M \models T'$, then there is a model M' of T s.t. $M' \sim_\mu M$.

Sound forgetting is weaker than forgetting, which means that its results may contain spurious models compared with that of forgetting. Conversely, complete forgetting is stronger, meaning that its results contain fewer models. The relation among forgetting, sound forgetting and complete forgetting can be summarized as follows:

Proposition 4 *Let $Mod(T)$ be the set of models of theory T, and $T' \in sforget(T; \mu)$ and $T'' \in cforget(T; \mu)$. Then*

$$Mod(T') \supseteq Mod(forget(T; \mu)) \supseteq Mod(T'').$$

Since orders of forgotten symbols do not affect results of forgetting, if P is a set of symbols, we use $forget(T; P)$ to denote forgetting all the symbols μ ($\mu \in P$) from theory T. Similar notations are $cforget(T; P)$ and $sforget(T; P)$.

Next, we introduce *mapping axioms* to indicate relations between low-level symbols and high-level ones. A set of mapping axioms can be viewed as an axiomatization of the m-bisimulation relation specified by a refinement mapping.

We first introduce a new predicate $\mathbb{B}$ over situation pairs. Then mapping axioms specify $\mathbb{B}$ as follows:

Definition 14 Given a refinement mapping m, mapping axioms $\mathcal{D}_m$ consist of

- $\mathbb{B}(S_0, S_0)$;
- for any high-level fluent $\mathbf{F} \in \mathcal{F}_h$,
  $$\mathbb{B}(s_1, s_2) \supset \mathbf{F}(\vec{x}, s_1) \equiv m(\mathbf{F})(\vec{x}, s_2);$$
- for any high-level action $\mathbf{A} \in \mathcal{A}_h$,
  $$\mathbb{B}(s_1, s_2) \wedge Do(\mathbf{A}(\vec{x}), s_1, s_1') \supset \exists s_2'.Do(m(\mathbf{A})(\vec{x}), s_2, s_2') \wedge \mathbb{B}(s_1', s_2');$$
  $$\mathbb{B}(s_1, s_2) \wedge Do(m(\mathbf{A})(\vec{x}), s_2, s_2') \supset \exists s_1'.Do(\mathbf{A}(\vec{x}), s_1, s_1') \wedge \mathbb{B}(s_1', s_2').$$

With all the above, we show that sound abstractions can be obtained via sound forgetting all low-level symbols from the low-level action theory together with the mapping axioms for m-bisimulation. Let $\mathcal{D}'_{una}$ be the unique name axioms for both high-level actions and low-level actions, indicating that high-level and low-level actions all stand for different actions. Formally speaking,

Theorem 2 *Given a low-level theory $\mathcal{D}_l$ and an NDU refinement mapping m, if*

$$T \in sforget(\mathcal{D}_l \cup \mathcal{D}'_{una} \cup \mathcal{D}_m; \mathcal{A}_l \cup \mathcal{F}_l \cup \{\mathbb{B}\}),$$

then T is a sound abstraction of $\mathcal{D}_l$.

Proof: ($sketch$) According to sound abstraction, we prove that for any model M_1 of $\mathcal{D}_l$, there is a model M_2 of T s.t. $M_2 \sim_m M_1$. We first construct a structure M_2 from M_1 by extending M_1 to M_1' s.t. $M_1' \models \mathcal{D}_l \cup \mathcal{D}'_{una} \cup \mathcal{D}_m$ and then letting $M_2 \sim_{\mathcal{A}_l \cup \mathcal{F}_l \cup \{\mathbb{B}\}} M_1'$. According to Def. 12 (sound forgetting), it follows that $M_2 \models T$. We then show $M_2 \sim_m M_1$ as follows: $M_1' \sim_m M_1$ is trivial since $M_1' \models \mathcal{D}_l \cup \mathcal{D}_m$; because M_1' and M_2 agree on all the valuations of high-level elements, it follows that $M_2 \sim_m M_1$. ■

Similarly, complete abstractions can be obtained via complete forgetting as follows.

Theorem 3 *Given a low-level theory $\mathcal{D}_l$ and an NDU refinement mapping m, if*

$$T \in cforget(\mathcal{D}_l \cup \mathcal{D}'_{una} \cup \mathcal{D}_m; \mathcal{A}_l \cup \mathcal{F}_l \cup \{\mathbb{B}\}),$$

then T is a complete abstraction of $\mathcal{D}_l$.

Proof: The proof is similar to the proof of Theorem 2. ■

According to Theorem 2, Theorem 3 and Prop. 4, we have:

Corollary 1 *Given a low-level theory $\mathcal{D}_l$ and an NDU refinement mapping m, we have that*

$$forget(\mathcal{D}_l \cup \mathcal{D}'_{una} \cup \mathcal{D}_m; \mathcal{A}_l \cup \mathcal{F}_l \cup \{\mathbb{B}\})$$

is a sound and complete abstraction of $\mathcal{D}_l$.

It follows that NDU is sufficient for the existence of a sound and complete abstraction for a mapping. Note that our theorems characterize the semantics of abstractions independently of their form, which is that of complex second-order theories, and which may not be representable as any kind of BAT. In the next section, we identify a further condition on refinement mappings that ensure the latter.

4 Computing GBAT Abstractions via Forgetting

In this section, we investigate restrictions that are necessary and sufficient for obtaining correct abstractions in the form of generalized BATs (GBATs). Moreover, we show how to compute GBAT abstractions under such a restriction, by extending regression and progression. Finally, we show that in the propositional case, GBAT abstractions are always computable under the restriction, and show how to decide whether a candidate mapping satisfies the restriction.

4.1 Markovian Refinement Mappings

BATs have the Markov property, which means that the executability conditions and the effects of actions are fully determined by the present state of the system [8]. To ensure that the abstraction we obtain is Markovian, in the following, we define the Markovian restriction for a refinement mapping given a low-level BAT.

First, we say that two situations are *in the same state* if they evaluate all the fluents the same. Formally speaking,

Definition 15 In a situation calculus language $\mathcal{L}$, we say that two situation terms σ_1 and σ_2 are in the same state under two structures M_1 and M_2 with the same object domain, written $\sigma_1 \approx_{\mathcal{L}}^{M_1,M_2} \sigma_2$, if for any fluent F in $\mathcal{L}$ and any variable assignment v,

$$M_1, v \models F(\vec{x}, \sigma_1) \text{ iff } M_2, v \models F(\vec{x}, \sigma_2).$$

With a refinement mapping m, one can say that two low-level situation terms σ_1 and σ_2 are *in the same abstract state*, denoted as $\sigma_1 \approx_{m(\mathcal{L})}^{M_1,M_2} \sigma_2$, if they evaluate the refinement of any high-level fluent $\mathbf{F}$, i.e., low-level formula $m(\mathbf{F})$ the same.

Based on this concept, we say that a formula is Markovian if its truth value only depends on the current state of the system, but not on situations which store history information. Formally,

Definition 16 In a situation calculus language $\mathcal{L}$, we say that a situation calculus formula $\varphi(s)$ whose only free variable is s, is Markovian iff given any structures M_1, M_2 and situation terms σ_1, σ_2, if σ_1, σ_2 are in the same state wrt M_1, M_2, then

$$M_1 \models \varphi(\sigma_1) \text{ iff } M_2 \models \varphi(\sigma_2).$$

Note that Markovian formulas can be second-order. Based on the notion of Markovian formulas, we extend BATs to generalized BATs.

Definition 17 A generalized BAT (GBAT) has the same form as a BAT except that formulas in $\mathcal{D}_{S_0}$, and the right hand side of formulas in precondition axioms and SSAs, are Markovian formulas.

We first illustrate that given a refinement mapping for a low-level theory, correct abstractions may go beyond the form of GBATs.

Example 5 Consider the blocks world with the initial database: $D_{S_0} = \{ontable(B_1, S_0), ontable(B_2, S_0) \vee on(B_2, B_1, S_0)\}$ We provide a refinement mapping m: high-level fluent *hontable* denotes the property that there exists one block on the table; high-level action *hmtb* means the same as low-level action *movetotable*. That is

$$m(hontable) = \exists x.ontable(x),$$
$$m(hmtb)(x) = movetotable(x).$$

We show that the precondition axiom for high-level action $hmtb(x)$ cannot be any Markovian formula. Consider two low-level models

M_1, M_2 with object domain $\{B_1, B_2\}$. In the initial situation, we only know that B_1 is on the table. As for block B_2, it is on B_1 in M_1, while it is on the table in M_2. To construct a sound abstraction, there must be M_3, M_4 such that $M_3 \sim_m M_1$ and $M_4 \sim_m M_2$. It follows that both $M_3 \models hontable(S_0)$ and $M_4 \models hontable(S_0)$, which means $S_0^{M_3}$ and $S_0^{M_4}$ are in the same state. But $M_3 \models Poss(hmtb(B_2), S_0)$ and $M_4 \not\models Poss(hmtb(B_2), S_0)$, which violates the Markov property.

In the following, we introduce the notion of $Markovian$ $refinement$ $mappings$. We first define the *m-reachable situations* of a low-level action theory given a refinement mapping m. Here m-reachable situations are situations needed to form an m-bisimilar relation, i.e., situations that need to be abstracted. They can be formalized by using notations in [1]: given a refinement mapping m, let π_m be a low-level program that characterizes doing any high-level action at the low-level, i.e., $\pi_m \doteq [\pi \vec{x}_1.m(\mathbf{A_1})(\vec{x_1}) \mid \ldots \mid \pi \vec{x}_n.m(\mathbf{A_n})(\vec{x_n})]$ for $\mathbf{A_i} \in \mathcal{A}_h$; then π_m^* represents doing any high-level action sequence at the low-level; so the set of m-reachable situations, written $m\text{-}reachable(s)$, can be expressed as: $m\text{-}reachable(s) \doteq Do(\pi_m^*, S_0, s)$.

We then introduce the Markovian restriction on mappings at the low level. Intuitively, it requires that for any two m-reachable situations, if they are in the same abstract (high-level) state, then for any high-level action $\mathbf{A}$, the executability condition and action effects are $indistinguishable$ in these two situations. It follows that the executability conditions and action effects are describable using Markovian formulas over the high-level language. Formally, we have:

Definition 18 Given a low-level action theory $\mathcal{D}_l$ and a refinement mapping m, we say that *m is Markovian wrt $\mathcal{D}_l$*, if for any models M_1, M_2 of $\mathcal{D}_l$, and for any ground situation terms σ_1, σ_2 such that $M_1 \models m\text{-}reachable(\sigma_1)$ and $M_2 \models m\text{-}reachable(\sigma_2)$ and σ_1, σ_2 are in the same abstract state, then for any high-level action $\mathbf{A}$, any high-level fluent $\mathbf{F}$ and any variable assignment v, we have

- $M_1, v \models \exists s' Do(m(\mathbf{A})(\vec{x}), \sigma_1, s')$ iff
  $M_2, v \models \exists s' Do(m(\mathbf{A})(\vec{x}), \sigma_2, s')$,
- $M_1, v \models \exists s' Do(m(\mathbf{A})(\vec{x}), \sigma_1, s') \wedge m(\mathbf{F})(\vec{y}, s')$ iff
  $M_2, v \models \exists s' Do(m(\mathbf{A})(\vec{x}), \sigma_2, s') \wedge m(\mathbf{F})(\vec{y}, s')$.

Proposition 5 *Given a low-level action theory $\mathcal{D}_l$ and a refinement mapping m, if there exists a sound and complete abstraction $\mathcal{D}_h$ of the form GBAT, then m is Markovian.*

Proof: It is proved by contradiction. Assume that the mapping is not Markovian and suppose it violates Item 1 in Def. 18. We know that there are two m-reachable situations s_1 and s_2 that are in the same high-level state but have different executability wrt $m(A)(\vec{t})$. It follows that there are two m-isomorphic high-level situations s_1^h and s_2^h that are in the same state. But the evaluation on the executability of $A(\vec{t})$ in s_1^h and s_2^h yields a contradiction: since two situations are in the same state, it follows that $Poss(A(\vec{t}), s_1^h) \equiv Poss(A(\vec{t}), s_2^h)$; according to the definition of m-bisimulation, we have that $Poss(A(\vec{t}), s_1^h) \not\equiv Poss(A(\vec{t}), s_2^h)$. The case where Item 2 is violated can be handled similarly ■

Since the Markovian restriction is necessary for a sound and complete abstraction, theoretically, it is implied by the results for verifying sound abstractions and complete abstractions in the agent abstraction framework [1]. Next, we identify it as a sufficient condition.

4.2 Computing Abstractions via Forgetting

We first define progression wrt programs to compute m-reachable situations. Let $\phi[x/y]$ denote replacing every x in ϕ with y.

Definition 19 Given situation-suppressed formula ϕ and Golog program δ, we define extended progression $prog[\phi(s), \delta]$ inductively:

- $prog[\phi(s), \alpha] \doteq \mathcal{P}_{\mathcal{D}}[\Pi_\alpha(s) \wedge \phi(s), \alpha][do(\alpha, s)/s]$, where $\Pi_\alpha(s)$ is a formula such that $Poss(\alpha, s) \equiv \Pi_\alpha(s)$;
- $prog[\phi(s), \psi?] \doteq \psi[s] \wedge \phi(s)$;
- $prog[\phi(s), \delta_1; \delta_2] \doteq prog[prog[\phi(s), \delta_1], \delta_2]$;
- $prog[\phi(s), \delta_1 | \delta_2] \doteq prog[\phi(s), \delta_1] \vee prog[\phi(s), \delta_2]$;
- $prog[\phi(s), (\pi x)\delta(x)] \doteq (\exists x) prog[\phi(s), \delta(x)]$;
- $prog[\phi(s), \delta^*] \doteq [\mathbf{lfp}_{Z,s} \phi(s) \vee prog[Z(s), \delta]](s)$.

Progressing a formula $\phi(s)$ wrt a program δ results in a state formula which represents all the reachable situations resulting from executing program δ in a certain situation satisfying ϕ.

Proposition 6 Given a basic action theory $\mathcal{D}$, a Golog program δ and a situation-suppressed formula ϕ, we have:

$$\mathcal{D} \models prog[\phi(s), \delta] \equiv \exists s'. \phi[s'] \wedge Do(\delta, s', s).$$

Proof: $(sketch)$ We prove it by structural induction. The base case $\delta \doteq \alpha$ is proved by Theorem 1 and the definition of progression; for the case $\delta \doteq \delta^*$, we prove that the semantics of least fixed point logic is equivalent to the semantics of Do by induction on the number of iterations for constructing the inductive fixed point. ■

Proposition 7 *$prog[\phi(s), \delta]$ is a Markovian formulas.*

Proof: $(sketch)$ We prove it by structural induction. The base case $\delta \doteq \alpha$ is proved by the definition of progression. When proving the case $\delta \doteq \delta^*$, we prove it by induction on the number of iterations of the inductive semantics of LFP. ■

Note that $prog[\phi(s), \delta]$ is generally second-order, and is not guaranteed to terminate for the construction of an inductive fixed point when mentioning δ^*.

We then have $m\text{-}reachable(s) \equiv prog[Init(s), \pi_m^*]$, where $Init(s) \doteq \bigwedge \mathcal{D}_{S_0}[S_0/s]$. We illustrate this by an example.

Example 3 cont'd In the blocks world, we restrict the number of objects to be finite via adding domain closure axiom $\mathcal{K}$, i.e., $\forall x.[x = B_1 \vee x = B_2]$. We add $\mathcal{D}_{una}$ implicitly as a component for simplification. Let mtb be short for $movetotable$ and $ontb$ for $ontable$. We first show the result of progression wrt $Init(s)$ and $(\pi x) mtb(x)$:

$$\begin{aligned} & prog[Init(s), (\pi x) mtb(x)] \\ \equiv & \exists x \exists \vec{R}. \{Init(s) \wedge \bigwedge \mathcal{D}_{ss}[mtb(x), s] \uparrow s\}[do(mtb(x), s)/s]. \end{aligned}$$

With domain closure axiom $\mathcal{K}$, the above result can be reduced to $\forall x.ontable(x, s) \wedge \forall x.clear(x, s) \wedge \forall x, y. \neg on(x, y, s)$, as in the initial situation, block B_1 is on the table and block B_2 is on B_1, and all the blocks are on the table after moving one block to the table.

If all the blocks are on the table, the action to move any block to the table is no longer executable. So we have:

$$prog[Init(s), (\pi x) mtb(x); (\pi x) mtb(x)] \equiv \bot.$$

Based on above results, we know that $prog[Init(s), [(\pi x)mtb(x)]^*]$

$$\begin{aligned}&\equiv [\mathbf{lfp}_{Z,s} Init(s) \lor prog[Z(s), (\pi x)mtb(x)]](s)\\ &\equiv Init(s) \lor prog[Init(s), (\pi x)mtb(x)]\\ &\quad \lor prog[Init(s), (\pi x)mtb(x); (\pi x)mtb(x)] \ldots\\ &\equiv Init(s) \lor prog[Init(s), (\pi x)mtb(x)].\end{aligned}$$

For high-level action $moveany$, $prog[Init(s), m(moveany)]$

$$\begin{aligned}&\equiv prog[Init(s), [(\pi x)mtb(x)]^*; \forall x.ontb(x)?]\\ &\equiv prog[Init(s) \lor prog[Init(s), (\pi x)mtb(x)], \forall x.ontb(x)?]\\ &\equiv prog[Init(s), (\pi x)mtb(x)].\end{aligned}$$

Therefore, the result of computing $m\text{-}reachable(s)$ is as follows:

$$\begin{aligned}&\equiv prog[Init(s), [m(moveany)]^*]\\ &\equiv Init(s) \lor prog[Init(s), m(moveany)]\\ &\equiv Init(s) \lor prog[Init(s), (\pi x)mtb(x)].\end{aligned}$$

To compute state descriptions for executability conditions and effects, here we present existentially extended regression, compared to universally extended regression introduced in [13]. Notation $\mathcal{R}[\phi(s), \delta]$ denotes a state formula expressing that there exists an execution of program δ starting from s and making ϕ hold.

Definition 20 Given a situation-suppressed formula ϕ and a Golog program δ, we define the extended regression $\mathcal{R}[\phi(s), \delta]$ inductively:

- $\mathcal{R}[\phi(s), \alpha] \doteq \mathcal{R}_D[Poss(\alpha, s) \land \phi(do(\alpha, s))]$;
- $\mathcal{R}[\phi(s), \psi?] \doteq \psi[s] \land \phi(s)$;
- $\mathcal{R}[\phi(s), \delta_1; \delta_2] \doteq \mathcal{R}[\mathcal{R}[\phi(s), \delta_2], \delta_1]$;
- $\mathcal{R}[\phi(s), \delta_1 | \delta_2] \doteq \mathcal{R}[\phi(s), \delta_1] \lor \mathcal{R}[\phi(s), \delta_2]$;
- $\mathcal{R}[\phi(s), (\pi x)\delta(x)] \doteq (\exists x)\mathcal{R}[\phi(s), \delta(x)]$;
- $\mathcal{R}[\phi(s), \delta^*] \doteq [\mathbf{lfp}_{Z,s} \phi(s) \lor \mathcal{R}[Z(s), \delta]](s)$.

Proposition 8 Given a basic action theory $\mathcal{D}$, a Golog program δ and a situation-suppressed formula ϕ, we have:

$$\mathcal{D} \models \mathcal{R}[\phi(s), \delta] \equiv \exists s'.Do(\delta, s, s') \land \phi[s'].$$

Proof: It is similar to the proof of Prop. 6 except that the base case is proved by the definition of one-step regression. ∎

Proposition 9 *$\mathcal{R}[\phi(s), \delta]$ is a Markovian formulas.*

Proof: It is similar to the proof of Prop. 7. ∎

Example 3 cont'd With the domain closure axiom $\mathcal{K}$, given

$$\begin{aligned}&\mathcal{R}[\forall x.ontb(x,s), [(\pi x)mtb(x)]^*]\\ \equiv&\mathbf{lfp}_{Z,s}[\forall x.ontb(x,s) \lor \mathcal{R}[Z(s), (\pi x)mtb]](s)\\ \equiv&\forall x.ontb(x,s) \lor \mathcal{R}[\forall x.ontb(x,s), (\pi x)mtb]\\ &\quad \lor \mathcal{R}[\forall x.ontb(x,s), (\pi x)mtb; (\pi x)mtb] \ldots\\ \equiv&\forall x.ontb(x,s) \lor \mathcal{R}[\forall x.ontb(x,s), (\pi x)mtb]\\ \equiv&\forall x.ontb(x,s) \lor \exists y.clear(y,s) \land \neg ontb(y,s)\\ &\quad \land \forall x.[x = y \lor ontb(x,s)] \qquad (1)\end{aligned}$$

(note that if without $\mathcal{K}$, the result is second-order), we have:

$$\begin{aligned}&\mathcal{R}[\top, m(moveany)]\\ \equiv&\mathcal{R}[\top, [(\pi x)mtb(x)]^*; \forall x.ontb(x)?]\\ \equiv&\mathcal{R}[\forall x.ontb(x,s), [(\pi x)mtb(x)]^*].\end{aligned}$$

To compute the result (1), intuitively, the regression of $\forall x.ontb(x,s)$ wrt $(\pi x)mtb(x)$ results in a formula ϕ, stating that there is one block not on the table. And the regression of ϕ wrt $(\pi x)mtb(x)$ results in two blocks not on the table. Since the domain is restricted to have two blocks, only one of the blocks can be not on the table.

With the extensions, we show how to construct a correct abstraction of the form GBAT. Let Φ_m be the relation between high-level fluents and low-level fluents, defined as $\bigwedge_{\mathbf{F} \in \mathcal{F}_h} \mathbf{F}(\vec{x}) \equiv m(\mathbf{F})(\vec{x})$.

Theorem 4 *Given a low-level BAT $\mathcal{D}_l$ and a Markovian refinement mapping m, let T be $\Sigma \cup \mathcal{D}^h_{una} \cup \mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ap} \cup \mathcal{D}^h_{ss}$, where*

- *$\mathcal{D}^h_{S_0} \doteq forget(\mathcal{D}_{S_0} \cup \Phi_m[S_0]; \mathcal{F}_l)$;*
- *$\mathcal{D}^h_{ap}$ contains the set of sentences: for any $\boldsymbol{A} \in \mathcal{A}_h$,*
  $Poss(\boldsymbol{A}(\vec{x}), s) \equiv forget(prog[Init(s), \pi^*_m] \land \mathcal{R}[\top(s), m(\boldsymbol{A})(\vec{x})] \land \Phi_m[s]; \mathcal{F}_l)$;
- *$\mathcal{D}^h_{ss}$ contains the set of sentences: for any $\boldsymbol{A} \in \mathcal{A}_h$ and $\boldsymbol{F} \in \mathcal{F}_h$,*
  $\boldsymbol{F}(\vec{y}, do(\boldsymbol{A}(\vec{x}), s)) \equiv forget(prog[Init(s), \pi^*_m] \land \mathcal{R}[\boldsymbol{F}(\vec{y}, s), m(\boldsymbol{A})(\vec{x})] \land \Phi_m[s]; \mathcal{F}_l)$.

Then T is a sound and complete abstraction of $\mathcal{D}_l$.

Proof: ($sketch$) On one side, we prove that T is a sound abstraction. Firstly, for any model M_l of $\mathcal{D}_l$, we construct a structure M_h as follows: we first construct a substructure M' from M_l s.t. it only contains fluents in the initial situation, and we expand M' to satisfy $\Phi_m[S_0]$; we then construct M'' s.t. $M'' \sim_{\mathcal{F}_l} M'$, and it follows $M'' \models \mathcal{D}^h_{S_0}$; we then obtain M_h by extending M'' to satisfy $\Sigma \cup \mathcal{D}^h_{una} \cup \mathcal{D}^h_{ap} \cup \mathcal{D}^h_{ss}$. Secondly, we prove $M_h \sim_m M_l$ by induction on the length of action sequence $\vec{\alpha}$: The base case (S_0) is straightforward with properties of the structures due to the construction. Induction: suppose $S^h_n \sim^{M_h, M_l}_m S^l_n$, where S^h_n (resp. S^l_n) results from executing high-level (resp. the refinement of high-level) actions of n-length, we prove that they have the same executability and effects for any high-level action. We prove the executability as follows: for any $\mathbf{A} \in \mathcal{A}_h$ and variable assignment v, we prove that $M_l, v \models \exists s.Do(m(\mathbf{A})(\vec{x}), S^l_n, s)$ iff $M_h, v \models Poss(\mathbf{A}(\vec{x}), S^h_n)$. It follows that we prove $M_l, v \models \exists s.Do(m(\mathbf{A})(\vec{x}), S^l_n, s)$ iff $\exists M \exists S$ such that $S^h_n \sim^{M_h, M}_m S$ and $M, v \models \mathcal{D}_{S_0} \land Do(\pi^*_m, S_0, S) \land \exists s.Do(m(\mathbf{A})(\vec{x}), S, s)$ (derived from the right hand side of $Poss(\mathbf{A}(\vec{x}), S^h_n)$ with Def. 4, Prop. 6 and Prop. 8). Proving forth is trivial, and we prove back with the property of Markovian mappings (Def. 18). Proving the effects and the other side (complete abstraction) is similar, so we omit them here. ∎

The above theorem says that, to construct the precondition axiom for high-level action $\mathbf{A}$, we first compute a state description for all m-reachable situations via extended progression; then we construct a state description for the executability condition via extended regression; and we finally obtain a high-level state description for $\mathbf{A}$ via forgetting all low-level fluents from the conjunction of the two state descriptions together with the fluent relation between the two-levels. The construction of action effects is similar.

Example 3 cont'd For the executability condition for the high-level action $moveany$, remember that we have computed $prog[Init(s), [m(moveany)]^*]$ and $\mathcal{R}[\top, m(moveany)]$. So

$$\begin{aligned}&Poss(moveany, s)\\ \equiv&forget(prog[Init(s), [m(moveany)]^*] \land \Phi_m[s]\\ &\land \mathcal{R}[\top(s), m(moveany)]; \{ontable(x,s), clear(x,s), on(x,y,s)\})\\ \equiv&\top \text{ (after simplification)},\end{aligned}$$

which means $moveany$ is always executable. Note that a program such as "moving all the blocks to the table", which we abstract to high-level action $moveany$, might be a useful abstract step to achieve a goal. We could generate a plan in the high-level theory to achieve a goal, and then perform further planning at the low level to refine the high-level plan to a complete low-level plan.

4.3 Computability in the Propositional Case

We show that abstractions of the form GBATs are computable in the propositional case.

Definition 21 A BAT is *propositional definable* (P-definable) if formulas in $\mathcal{D}_{S_0}, \mathcal{D}_{ap}$ and $\mathcal{D}_{ss}$ are P-definable.

In the propositional case, the state space of $\mathcal{D}$ is finite. It is easy to show that $\mathcal{R}[\phi(s), \delta]$ and $prog[\phi(s), \delta]$ are P-definable and computable. Further, forgetting is P-definable and computable. It follows that GBAT abstractions are P-definable and computable.

Theorem 5 *Given a Markovian refinement mapping m and a low-level action theory $\mathcal{D}_l$, if $\mathcal{D}_l$ is P-definable, then a sound and complete abstraction of the form GBAT is P-definable and computable.*

4.4 Verifying that a Mapping is Markovian

We show how to decide whether a candidate mapping is Markovian by using elements of Theorem 4. For any high-level action $\mathbf{A}(\vec{t})$, we check the requirement for the executability condition by checking the unsatisfiability of the following formula:

$$forget(prog[Init(s), \pi_m^*] \wedge \mathcal{R}[\top(s), m(\mathbf{A})(\vec{t})] \wedge \Phi_m[s]; \mathcal{F}_l)$$
$$\wedge\, forget(prog[Init(s'), \pi_m^*] \wedge \neg\mathcal{R}[\top(s'), m(\mathbf{A})(\vec{t})] \wedge \Phi_m[s']; \mathcal{F}_l).$$

If the above formula is satisfiable, then the candidate mapping is not Markovian, since it follows that there are two low-level situations that (via forgetting) are in the same abstract state and have different executability conditions. Checking the requirement for action effects is similar, so we omit the details here.

5 Conclusion

In this paper, we have shown that given a low-level action theory and a refinement mapping: firstly, how one can characterize the semantics of a correct abstraction via forgetting under the non-deterministic uniform condition; secondly, how one can compute a correct abstraction of the form generalized BATs under the Markovian restriction. Moreover, we have shown that in the propositional case, the abstractions are always computable. Our method should be adaptable to other dynamic domain/agent modelling frameworks, such as "action languages", PDDL planning domain descriptions, etc.

A limitation of our current approach is the restriction to theories that are Markovian, and to deterministic primitive actions that should match suitable programs when programs involve non-determinism. Extensions to non-Markovian theories and non-deterministic primitive actions are interesting research directions. In the future, we would also like to investigate sufficient conditions under which correct abstractions are always first-order-definable. Furthermore, based on the presented work, we would like to explore applying the agent abstraction framework to planning by providing a mapping as a guide. The idea is to first generate a plan in the constructed high-level theory, and then perform further planning at the low level to refine the high-level plan to a complete low-level plan. Also, we would like to study (partially) automated generation of these mappings by integrating counterexample-guided synthesis.

Acknowledgments

We thank the anonymous reviewers for helpful comments. We acknowledge support from the Natural Science Foundation of China under Grant No. 61572535 and the National Science and Engineering Research Council of Canada.

References

[1] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance, 'Abstraction in situation calculus action theories', in *AAAI*, (2017).
[2] Blai Bonet, Giuseppe De Giacomo, Hector Geffner, and Sasha Rubin, 'Generalized planning: Non-deterministic abstractions and trajectory constraints', in *IJCAI*, pp. 873–879, (2017).
[3] Edmund M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, 'Counterexample-guided abstraction refinement', in *CAV*, (2000).
[4] Edmund M. Clarke, Orna Grumberg, and David E. Long, 'Model checking and abstraction', *ACM Trans. Program. Lang. Syst.*, **16**(5), 1512–1542, (1994).
[5] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, 'Congolog, a concurrent programming language based on the situation calculus', *Artif. Intell.*, **121**(1-2), 109–169, (2000).
[6] Thomas Eiter and Gabriele Kern-Isberner, 'A brief survey on forgetting from a knowledge representation and reasoning perspective', *KI*, **33**(1), 9–33, (2019).
[7] Cormac Flanagan and Shaz Qadeer, 'Predicate abstraction for software verification', in *POPL*, pp. 191–202, (2002).
[8] Alfredo Gabaldon, 'Non-markovian control in the situation calculus', *Artif. Intell.*, **175**(1), 25–48, (2011).
[9] Ricardo Gonçalves, Matthias Knorr, and João Leite, 'The ultimate guide to forgetting in answer set programming', in *KR*, (2016).
[10] Malte Helmert, Patrik Haslum, and Jörg Hoffmann, 'Flexible abstraction heuristics for optimal sequential planning', in *ICAPS*, (2007).
[11] Craig A. Knoblock, 'Automatically generating abstractions for planning', *Artif. Intell.*, **68**(2), 243–302, (1994).
[12] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *J. Log. Program.*, **31**(1-3), 59–83, (1997).
[13] Naiqi Li and Yongmei Liu, 'Automatic verification of partial correctness of golog programs', in *IJCAI*, pp. 3113–3119, (2015).
[14] Fangzhen Lin and Ray Reiter, 'Forget it!', in *Working Notes of AAAI Fall Symposium On Relevance AAAI, Menlo Park, CA, 1994*, (1994).
[15] Fangzhen Lin and Raymond Reiter, 'How to progress a database', *Artif. Intell.*, **92**(1-2), 131–167, (1997).
[16] John McCarthy and Patrick J. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', *Machine Intelligence*, **4**, 464–502, (1969).
[17] Robin Milner, 'An algebraic definition of simulation between programs', in *IJCAI*, pp. 481–489, (1971).
[18] Peiming Mo, Naiqi Li, and Yongmei Liu, 'Automatic verification of golog programs via predicate abstraction', in *ECAI*, (2016).
[19] Raymond Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.
[20] John K. Slaney and Sylvie Thiébaux, 'Blocks world revisited', *Artif. Intell.*, **125**(1-2), 119–153, (2001).
[21] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein, 'Learning generalized plans using abstract counting', in *AAAI*, (2008).
[22] Alfred Tarski, 'A lattice-theoretical fixpoint theorem and its applications', *Pacific Journal of Mathematics*, **5**(2), 285–309, (1955).
[23] Kewen Wang, Abdul Sattar, and Kaile Su, 'A theory of forgetting in logic programming', in *AAAI*, pp. 682–688, (2005).
[24] Yan Zhang and Yi Zhou, 'Knowledge forgetting: Properties and applications', *Artif. Intell.*, **173**(16-17), 1525–1537, (2009).