# Hierarchical Agent Supervision

Bita Banihashemi
York University
Toronto, ON, Canada
bita@cse.yorku.ca

Giuseppe De Giacomo
Sapienza University of Rome
Rome, Italy
degiacomo@dis.uniroma1.it

Yves Lespérance
York University
Toronto, ON, Canada
lesperan@cse.yorku.ca

## ABSTRACT

Agent supervision is a form of control/customization where a supervisor restricts the behavior of an agent to enforce certain requirements, while leaving the agent as much autonomy as possible. To facilitate supervision, it is often of interest to consider hierarchical models where a high level abstracts over low-level behavior details. We study *hierarchical agent supervision* in the context of the situation calculus and the ConGolog agent programming language, where we have a rich first-order representation of the agent state. We define the constraints that ensure that the controllability of individual actions at the high level in fact captures the controllability of their implementation at the low level. On the basis of this, we show that we can obtain the maximally permissive supervisor by first considering only the high-level model and obtaining a high-level supervisor and then refining its actions locally, thus greatly simplifying the supervisor synthesis task.

## KEYWORDS

Reasoning about action, plans and change in multi-agent systems; Logics for agents and multi-agent systems

**ACM Reference Format:**
Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. 2018. Hierarchical Agent Supervision. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 10 pages.

## 1 INTRODUCTION

In many settings, we want to control an agent's behavior to conform to a set of specifications while preserving its autonomy as much as possible. The key challenge is then to synthesize the *maximally permissive supervisor* (MPS) that minimally constrains the behavior of the agent in the presence of uncontrollable actions so as to enforce the desired behavioral specifications. This problem is the central problem of Supervisory Control of Discrete Event Systems (SCDES), where it has been thoroughly studied in a finite-state setting [10, 34, 35]. Recently the problem has been lifted to a rich first-order state setting in [12] (referred as DLM in the following), where it has been studied in the context of the situation calculus [23, 29] and the ConGolog programming language [11].

To facilitate supervision, it is of interest to consider hierarchical models where a high level abstracts over low-level behavior details. This has already been considered in the finite state case in SCDES [34], but is even more critical in agents with complex first-order state representations. Exploiting the insights on abstraction in situation calculus action theories presented in [7], in this paper we extend the DLM framework to study *hierarchical agent supervision* in the context of the situation calculus and the ConGolog agent programming language. We assume that we have a low-level basic action theory and also a high-level basic action theory that abstracts over it. High-level fluents correspond to a state formula at the low level and high-level actions are associated with a ConGolog program that implements the action at the low level. Some of the actions at the low-level (and high-level) are uncontrollable, i.e., their occurrence cannot be prevented by the supervisor. Moreover, the behavior of the agent at the low level can be monitored at the high level, i.e., any complete low-level run of the agent must be a refinement of a sequence of high-level actions. The constraints on the agent's behavior to be enforced by the supervisor are represented by a high-level ConGolog program, which specifies the behaviors that are acceptable/desirable. Our task is to *synthesize a MPS for the low-level agent and specification* (which we can translate into a low-level program). We show that we can actually do this synthesis task by exploiting the high-level model, first obtaining a MPS at the high-level, and then refining its actions locally while remaining maximally permissive. Moreover, we show that this can be done incrementally, without precomputing the local refinements.

To allow this, we first identify the constraints required to ensure that controllability of individual actions at the high level accurately reflects the controllability of their refinements. Then we show that these constraints are in fact sufficient to ensure that any controllable set of runs at the high level has a controllable refinement that corresponds to it and vice versa. In particular, this applies to the MPS for any supervision specification represented by a high-level ConGolog program: the low-level MPS for the mapped specification is a refinement of the high-level MPS for the specification. Finally, as already mentioned, we also show that we can obtain the low-level MPS incrementally using the high-level MPS as a guide.

## 2 PRELIMINARIES

**Situation Calculus.** The *situation calculus* is a well known predicate logic language for representing and reasoning about dynamically changing worlds. All changes to the world are the result of *actions*, which are terms in the logic. A possible world history is represented by a term called a *situation*. The constant $S_0$ is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol $do$, such that $do(a, s)$ denotes the successor situation resulting from performing action $a$ in situation $s$. A precedence relation on situations $s$ and $s'$ denoted by $s \leq s'$ states that $s'$ is a successor situation of $s$ and that every action between $s$ and $s'$ is in fact executable. We write $do([a_1, a_2, \ldots, a_{n-1}, a_n], s)$ as an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s)) \ldots))$; for an action sequence $\vec{a}$,

*Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden.* 

we often write $do(\vec{a}, s)$ for $do([\vec{a}], s)$. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$). Within the language, one can formulate *basic action theories* (BATs) that describe how the world changes as a result of actions; see [29] for details of how these are defined. Hereafter, we will use $\mathcal{D}$ to refer to the BAT under consideration. We assume that there is a *finite number of action types* $\mathcal{A}$. Moreover, we assume that the terms of object sort are in fact a countably infinite set $\mathcal{N}$ of standard names for which we have the unique name assumption and domain closure. For simplicity, and w.l.o.g., we assume that there are no functions other than constants and no non-fluent predicates. A special predicate $Poss(a, s)$ is used to state that action $a$ is executable in situation $s$. The abbreviation $Executable(s)$ means that every action performed in reaching situation $s$ was executable in the situation in which it occurred.

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

$$\delta ::= \alpha \mid \varphi? \mid \delta_1; \delta_2 \mid \delta_1 | \delta_2 \mid \pi x.\delta \mid \delta^* \mid \delta_1 \| \delta_2 \mid \textbf{atomic}(\delta) \mid nil$$

In the above, $\alpha$ is an action term, possibly with parameters. $\varphi$ is a situation-suppressed formula, i.e., a formula with all situation arguments in fluents suppressed (also sometimes the situation argument is replaced by a placeholder *now*). As usual, we denote by $\varphi[s]$ the formula obtained from $\varphi$ by restoring the situation argument $s$ into all fluents in $\varphi$. The sequence of program $\delta_1$ followed by program $\delta_2$ is denoted by $\delta_1; \delta_2$. Program $\delta_1 | \delta_2$ allows for the nondeterministic choice between programs $\delta_1$ and $\delta_2$, while $\pi x.\delta$ executes program $\delta$ for *some* nondeterministic choice of a binding for object variable $x$ (observe that such a choice is, in general, unbounded). $\delta^*$ performs $\delta$ zero or more times. Program $\delta_1 \| \delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs $\delta_1$ and $\delta_2$, while $\textbf{atomic}(\delta)$ performs $\delta$ as an atomic unit, without allowing any interleaved actions [16]. Finally *nil* denotes the empty program, which requires to do nothing and is already terminated.

Formally, the semantics of ConGolog is specified in terms of single-step transitions, using the following two predicates [11]: *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program $\delta$ in situation $s$ may lead to situation $s'$ with $\delta'$ remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if an execution of $\delta$ can be considered completed in $s$. The definitions of $Trans$ and $Final$ we use are as in [15]; differently from [11], the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Predicate $Do(\delta, s, s')$ means that program $\delta$, when executed starting in situation $s$, has $s'$ as a terminating situation, 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 $Trans$. In the rest, we use $C$ to denote the axioms defining the ConGolog programming language.

**Situation-Determined Programs.** We use ConGolog programs to represent the possible behaviors of an agent. So, it is natural to assume that such programs are *situation-determined* (SD) [12], i.e., for every sequence of actions, the remaining program is uniquely determined by the resulting situation:

$$SituationDetermined(\delta, s) \doteq \forall s', \delta', \delta''. \\ Trans^*(\delta, s, \delta', s') \wedge Trans^*(\delta, s, \delta'', s') \supset \delta' = \delta'',$$

For example, program $(a; b) \mid (a; c)$ is not SD, while $a; (b \mid c)$ is (assuming the actions involved are always executable). Thus, a (partial) execution of a SD program is uniquely determined by the sequence of actions it has produced. Hence a program in a starting situation generates a set/language of action sequences, its executions, and operations like intersection and union become natural. Exploiting this, we will write $\mathcal{CR}_M(\delta, s)$ for the set of *complete runs* of program $\delta$ in situation $s$ in model $M$:

$$\mathcal{CR}_M(\delta, s) \doteq \{\vec{a} \mid M \models Do(\delta, s, do(\vec{a}, s))\}.$$

## 3 AGENT SUPERVISION

Agent supervision aims at restricting an agent's behavior to ensure that it conforms to a supervision specification while leaving it as much autonomy as possible. The objective is to customize the generic behavior of an existing agent, not to synthesize the agent from scratch. DLM assume that the agent's possible behaviors are represented by a (nondeterministic) SD ConGolog program $\delta^i$ relative to a BAT $\mathcal{D}$. The supervision specification is represented by another SD ConGolog program $\delta^s$. DLM extend ConGolog with two new constructs both of which preserve situation-determinateness: $\delta_1 \& \delta_2$ and $\textbf{set}(E)$. The former denotes the *intersection* or *synchronous concurrent execution* of programs $\delta_1$ and $\delta_2$. The latter denotes an infinitary nondeterministic branch; it takes an arbitrary set of sequences of actions $E$ and turns it into a program. $Trans$ and $Final$ for the new constructs are easily definable.[1]

Using the first construct it is straightforward to specify the result of supervision as the intersection of the agent and the specification processes ($\delta^i \& \delta^s$), but only if it is possible to control all the actions of the agent. However in general, some of agent's actions may be *uncontrollable*. These are often the result of interaction of an agent with external resources, or may represent aspects of agent's behavior that must remain autonomous and cannot be controlled directly. This is modeled by the special fluent $A_u(a, s)$ that means action $a$ is uncontrollable in situation $s$.

A supervision specification $\delta^s$ is defined to be *controllable* wrt the agent program $\delta^i$ in situation $s$ as follows:[2]

$$Controllable(\delta^s, \delta^i, s) \doteq \\ \forall s', a_u.(\exists s''. Do(\delta^s, s, s'') \wedge s \leq s' \leq s'') \wedge A_u(a_u, s') \wedge \\ (\exists s''. Do(\delta^i, s, s'') \wedge s < do(a_u, s') \leq s'') \supset \\ (\exists s''. Do(\delta^s, s, s'') \wedge s < do(a_u, s') \leq s'')$$

i.e., if we take an action sequence $\vec{a}$ that is a prefix of a complete run of $\delta^s$ and append to it an uncontrollable action $a_u$ such that $\vec{a}a_u$ is a prefix of a complete run of $\delta^i$, then $\vec{a}a_u$ must also be a prefix of a

[1] Specifically the axioms for the two constructs are:

$$Trans(\delta_1 \& \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta_1', s') \wedge Trans(\delta_2, s, \delta_2', s') \\ \wedge \delta' = \delta_1' \& \delta_2'$$
$$Final(\delta_1 \& \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$
$$Trans(\textbf{set}(E), s, \delta', s') \equiv \exists a, \vec{a}. a\vec{a} \in E \wedge Poss(a, s) \wedge \\ s' = do(a, s) \wedge \delta' = \textbf{set}(\{\vec{a} \mid a\vec{a} \in E \wedge Poss(a, s)\})$$
$$Final(\textbf{set}(E), s) \equiv \epsilon \in E$$

Thus $\delta_1 \& \delta_2$ executes only if both programs step to the same situation and is final when both programs are final, while $\textbf{set}(E)$ can execute any of the sequences of actions in $E$ and is final if $E$ includes the empty sequence of actions $\epsilon$.

[2] Our definition is equivalent to that of DLM, but the notation is clearer.

complete run of $\delta^s$.Also, we will write $Controllable_M(\delta^s, \delta^i, s)$ as an abbreviation for $M \models Controllable(\delta^s, \delta^i, s)$.

DLM define the *maximally permissive supervisor* (MPS) $mps(\delta^i, \delta^s, s)$ of the agent behavior $\delta^i$ which fulfills the supervision specification $\delta^s$ as:

$$mps(\delta^i, \delta^s, s) = \mathbf{set}(\bigcup_{E \in \mathcal{E}} E) \text{ where}$$
$$\mathcal{E} = \{E \mid \forall \vec{a} \in E.Do(\delta^i \,\&\, \delta^s, s, do(\vec{a}, s)) \wedge Controllable(\mathbf{set}(E), \delta^i, s)\}$$

i.e., the MPS is the union of all sets of action sequences that are complete runs of both $\delta^i$ and $\delta^s$ that are controllable wrt $\delta^i$ in situation $s$. In this definition, $Do$ is indirectly used to select the set $E$ of sequences of actions on which the $\mathbf{set}(E)$ construct is applied.

DLM show that their notion of MPS, $mps(\delta^i, \delta^s, s)$, has several nice properties: *i*) it always exists and is unique, *ii*) it is controllable wrt the agent behavior $\delta^i$ in $s$, *iii*) and it is the largest set of complete runs of $\delta^i$ that is controllable wrt $\delta^i$ in $s$ and satisfies the supervision specification $\delta^s$ in $s$, i.e., it is maximally permissive. Note that *mps* is an abbreviation which stands for the program obtained by using the set construct over a (possibly infinite) set of action sequences. This is more of a specification than a conventional program; DLM also introduce a special version of the synchronous concurrency construct ($\&_{A_u}$) that takes into account the fact that some actions are uncontrollable, which together with a form of lookahead search exactly captures the maximally permissive supervisor. In the remainder, for simplicity we just use $mps(\delta^i, \delta^s, s)$, but this more conventional alternative could be used instead.

## A Logistics Running Example

For our running example, we use a simple logistics domain. There is a shipment with ID 123 that is initially at a warehouse ($W$), and needs to be delivered to a Cafe ($Cf$), along a network of roads shown in Figure 1 (warehouse and cafe images are from freedesignfile.com).

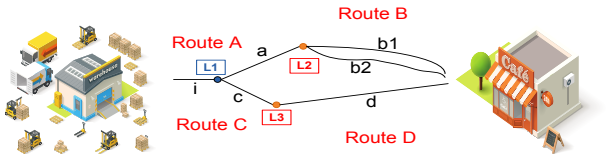


**Figure 1: Transport Logistics Example**

*BAT* $\mathcal{D}_l^{eg}$. We have an action $takeRoad(sID, t, o, d)$ for taking a shipment with ID *sID* from origin location $o$ to destination location $d$ along a road edge $t$, e.g., $takeRoad(123, Rd_i, W, L1)$ (we refer to road $x$ in Figure 1 as $Rd_x$). This action is executable when the agent is at location $o$ and road $t$ connects $o$ to $d$. The fluent $At_{LL}(sID, l, s)$ indicates that shipment *sID* is at location $l$ in $s$. $CnRoad(t, o, d, s)$ specifies the road connections in Figure 1. Also, $Rd_{b2}$ can only be used if the shipment travels during nighttime, represented by fluent $NT(sID, s)$. Performing delivery involves unloading the shipment ($unload(sID)$) and getting a signature ($getSignature(sID)$). $unload(sID)$ is executable when *sID* is at its destination (specified by the fluent $Dest_{LL}(sID, l, s)$), and $getSignature(sID)$ is executable when the shipment has already been unloaded. These actions are assumed to be *controllable*. We have two exogenous actions representing delays that may occur when the agent is at location $L3$: $delayBD(sID)$ due to bad weather and $delayRM(sID)$ due to road maintenance. These are the only *uncontrollable* actions. Finally, we have two types of express shipments: Express Same Day ($Exp1$) or Express 2 Days ($Exp2$).

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

$$Poss(takeRoad(sID, t, o, d), s) \equiv o \neq d \wedge At_{LL}(sID, o, s) \wedge CnRoad(t, o, d, s) \wedge (t = Rd_{b2} \supset NT(sID, s))$$
$$Poss(unload(sID), s) \equiv \exists l.Dest_{LL}(sID, l, s) \wedge At_{LL}(sID, l, s)$$
$$Poss(getSignature(sID), s) \equiv Unloaded(sID, s)$$
$$Poss(delayBD(sID), s) \equiv At_{LL}(sID, L3, s)$$
$$Poss(delayRM(sID), s) \equiv At_{LL}(sID, L3, s)$$

Moreover, $\mathcal{D}_l^{eg}$ includes the following SSAs:

$$At_{LL}(sID, l, do(a, s)) \equiv \exists l', r.a = takeRoad(sID, r, l', l) \vee At_{LL}(sID, l, s) \wedge \forall l', r.a \neq takeRoad(sID, r, l, l')$$
$$Unloaded(sID, do(a, s)) \equiv a = unload(sID) \vee Unloaded(sID, s)$$
$$Signed(sID, do(a, s)) \equiv a = getSignature(sID) \vee Signed(sID, s)$$
$$DelayedBD(sID, do(a, s)) \equiv a = delayBD(sID) \vee DelayedBD(sID, s)$$
$$DelayedRM(sID, do(a, s)) \equiv a = delayRM(sID) \vee DelayedRM(sID, s)$$

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

$\mathcal{D}_l^{eg}$ also contains the following initial state axioms:

$$\neg Exp1(sID, S_0), \quad Exp2(sID, S_0) \equiv sID = 123,$$
$$Dest_{LL}(sID, l, S_0) \equiv sID = 123 \wedge l = Cf,$$
$$At_{LL}(sID, l, S_0) \equiv sID = 123 \wedge l = W,$$
$$NT(sID, S_0) \equiv sID = 123,$$
$$A_l^u(a, S_0) \equiv \exists sID(a = delayBW(sID) \vee a = delayRM(sID))$$

together with a complete specification of $CnRoad$.

Suppose we have a supervision specification $\delta^l_{Spec}$ that says that any shipment that has been ordered, in our case just 123, must eventually be unloaded and signed for, and if it is an express shipment, it should never be delayed:

$$\delta^l_{Spec} = \&_{sID \in ShpOrd}[\pi a.a; ((Exp1(sID) \vee Exp2(sID)) \supset \neg(DelayedBD(sID) \vee DelayedRM(sID)))?]^*; (Unloaded(sID) \wedge Signed(sID))?$$

Intuitively, the MPS for this specification allows taking road $i$ followed by road $a$, and then taking either of roads $b1$ or $b2$, followed by unloading the shipment and getting customer's signature. Taking road $c$ is not allowed, as it may be followed by either of the uncontrollable actions $delayBW(sID)$ or $delayRM(sID)$, which are forbidden by the specification, as 123 is an express shipment.

Often, agents need to represent and reason with large amounts of knowledge about their environment and have complex behaviors. Due to complexity of the behavior logic, designing and enforcing specifications for the customization of the agent's behavior can be a difficult task. To facilitate this, we want to use *abstraction* and work with a simpler model of the agent. In the following sections, we will first develop a high-level model of our agent behavior/BAT. We will then provide the supervision specification in terms of the abstract model, and obtain a MPS for the abstract model of the agent. Finally, we will show how we can use this to obtain a MPS for the concrete/low-level model of the agent.

## 4 ABSTRACTING AGENT BEHAVIOR

In the agent abstraction framework of [7], there is a high-level (abstract) (HL) action theory $\mathcal{D}_h$ and a low-level (concrete) (LL) action theory $\mathcal{D}_l$ representing the agent's possible behaviors at different levels of detail. $\mathcal{D}_h$ (resp. $\mathcal{D}_l$) involves a finite set of primitive action types $\mathcal{A}_h$ (resp. $\mathcal{A}_l$) and a finite set of primitive fluent predicates $\mathcal{F}_h$ (resp. $\mathcal{F}_l$). Also, $\mathcal{D}_h$ and $\mathcal{D}_l$ are assumed to share no domain specific symbols except for the set of standard names for objects $\mathcal{N}$.

**Refinement Mapping**. To relate the two theories, a *refinement mapping* $m$ is defined as a function that associates each high-level primitive action type $A$ in $\mathcal{A}_h$ to a SD ConGolog program $\delta_A$ defined over the low-level theory that implements the action, i.e., $m(A(\vec{x})) = \delta_A(\vec{x})$; moreover, $m$ maps each situation-suppressed high-level fluent $F(\vec{x})$ in $\mathcal{F}_h$ to a situation-suppressed formula $\phi_F(\vec{x})$ defined over the low-level theory that characterizes the concrete conditions under which $F(\vec{x})$ holds in a situation. We extend the notation so that $m(\phi)$ stands for the result of substituting every fluent $F(\vec{x})$ in situation-suppressed formula $\phi$ by $m(F(\vec{x}))$. Also, we apply $m$ to action sequences with $m(\alpha_1, \ldots, \alpha_n) \doteq m(\alpha_1); \ldots; m(\alpha_n)$ for $n \geq 1$ and $m(\epsilon) \doteq nil$, where $\epsilon$ is the empty sequence of actions.

*m*-**Bisimulation**. To relate the high-level and low-level models/theories, a variant of bisimulation [26, 27] is defined as follows. Given $M_h$ a model of $\mathcal{D}_h \cup C$, and $M_l$ a model of $\mathcal{D}_l \cup C$, a relation $B \subseteq \Delta_S^{M_h} \times \Delta_S^{M_l}$ (where $\Delta_S^M$ stands for the situation domain of $M$) is an *m-bisimulation relation between* $M_h$ *and* $M_l$ if $\langle s_h, s_l \rangle \in B$ implies that: (*i*) $s_h \sim_m^{M_h, M_l} s_l$, i.e., $s_h$ and $s_l$ evaluate each high-level primitive fluent the same; (*ii*) for every high-level primitive action type $A$ in $\mathcal{A}_h$, if there exists $s_h'$ such that $M_h \models Poss(A(\vec{x}), s_h) \wedge s_h' = do(A(\vec{x}), s_h)$, then there exists $s_l'$ such that $M_l \models Do(m(A(\vec{x})), s_l, s_l')$ and $\langle s_h', s_l' \rangle \in B$; and (*iii*) for every high-level primitive action type $A$ in $\mathcal{A}_h$, if there exists $s_l'$ such that $M_l \models Do(m(A(\vec{x})), s_l, s_l')$, then there exists $s_h'$ such that $M_h \models Poss(A(\vec{x}), s_h) \wedge s_h' = do(A(\vec{x}), s_h)$ and $\langle s_h', s_l' \rangle \in B$. $M_h$ is *m-bisimilar* to $M_l$, written $M_h \sim_m M_l$, if and only if there exists an $m$-bisimulation relation $B$ between $M_h$ and $M_l$ such that $(S_0^{M_h}, S_0^{M_l}) \in B$.

*m*-**Refinement.** A *(ground low-level action sequence)* $\vec{a}$ *is an m-refinement of an executable (ground high-level action sequence)* $\vec{\alpha}$ *(wrt m-bisimilar models* $M_h \sim_m M_l$*)* if and only if $M_h \models Executable(do(\vec{\alpha}, S_0))$ and $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$.

**Sound abstractions.** In [7], $\mathcal{D}_h$ is a *sound abstraction of* $\mathcal{D}_l$ *relative to refinement mapping* $m$ if and only if, for all models $M_l$ of $\mathcal{D}_l \cup C$, there exists a model $M_h$ of $\mathcal{D}_h \cup C$ such that $M_h \sim_m M_l$. With a sound abstraction, whenever the high-level theory *entails* that a sequence of actions is executable and achieves a certain condition, then the low level must also entail that there exists an executable refinement of the sequence such that the "translated" condition holds afterwards. Moreover, whenever the low level considers the executability of a refinement of a high-level action is satisfiable, then the high level does also. A proof-theoretic characterization that provides the basis for automatically verifying that one has a sound abstraction is also given. Note that a dual notion is also defined: $\mathcal{D}_h$ is a *complete abstraction of* $\mathcal{D}_l$ *relative to refinement mapping* $m$ if and only if, for all models $M_h \cup C$ of $\mathcal{D}_h$, there exists a model $M_l$ of $\mathcal{D}_l \cup C$ such that $M_l \sim_m M_h$ (but we don't make use of this here).

**Example (cont.)** Returning to our running example, we define a high-level BAT that abstracts over some details of $\mathcal{D}_l^{eg}$.

*High-Level BAT* $\mathcal{D}_h^{eg}$. At the high level, we have abstract actions that represent choices of major routes, delivering a shipment, and an exogenous action representing a travel delay. Routes abstract over the roads; for instance, route $A$ is refined to road $i$ followed by road $a$, and route $B$ is refined to either road $b1$ or $b2$ (see Figure 1). The action $takeRoute(sID, r, o, d)$ can be performed to take shipment $sID$ from origin location $o$ to destination location $d$ via route $r$, e.g., $takeRoute(123, Rt_A, W, L2)$ (we refer to route $X$ in Figure 1 as $Rt_X$); it is executable when the shipment is initially at $o$ and route $r$ connects $o$ to $d$. Action $deliver(sID)$ abstracts over the unloading of the shipment and getting the customer's signature and can be performed to deliver shipment $sID$. It is executable when $sID$ is at its destination. Both of these actions are assumed to be *controllable*. $delay(sID)$ is an exogenous and *uncontrollable* action that may occur when the shipment is at location $L3$. This action abstracts over actions $delayBD(sID)$ and $delayRM(sID)$. Shipments may be high priority, represented by a fluent $Priority(sID, s)$.

$\mathcal{D}_h^{eg}$ includes the following precondition axioms:

$$
\begin{aligned}
&Poss(takeRoute(sID, r, o, d), s) \equiv o \neq d \wedge At_{HL}(sID, o, s) \wedge \\
&\quad CnRoute_{HL}(r, o, d, s) \\
&Poss(deliver(sID), s) \equiv \\
&\quad \exists l. Dest_{HL}(sID, l, s) \wedge At_{HL}(sID, l, s) \\
&Poss(delay(sID), s) \equiv At_{HL}(sID, L3, s)
\end{aligned}
$$

In the above, $CnRoute_{HL}(r, o, d, s)$ represents the routes in the map in Figure 1, $Dest_{HL}$ specifies the destination of the shipment and $At_{HL}$ indicates its location.

The high-level BAT also includes the following SSAs:

$$
\begin{aligned}
&At_{HL}(sID, l, do(a, s)) \equiv \exists l', r. a = takeRoute(sID, r, l', l) \vee \\
&\quad At_{HL}(sID, l, s) \ \wedge \forall l', r. a \neq takeRoute(sID, r, l, l') \\
&Delivered(sID, do(a, s)) \equiv a = deliver(sID) \vee Delivered(sID, s) \\
&Delayed(sID, do(a, s)) \equiv a = delay(sID) \vee Delayed(sID, s)
\end{aligned}
$$

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

$\mathcal{D}_h^{eg}$ contains the following initial state axioms:

$$
\begin{aligned}
&Priority(sID, S_0) \equiv sID = 123, \\
&Dest_{HL}(sID, l, S_0) \equiv sID = 123 \wedge l = Cf, \\
&At_{HL}(sID, l, S_0) \equiv sID = 123 \wedge l = W, \\
&A_h^u(a, S_0) \equiv \exists sID. a = delay(sID)
\end{aligned}
$$

together with a complete specification of $CnRoute_{HL}(r, l_s, l_e, S_0)$.

*Refinement Mapping* $m^{eg}$. We specify the relationship between the high-level and low-level BATs through a refinement mapping $m^{eg}$, which is defined as follows:

$$
\begin{aligned}
&m^{eg}(takeRoute(sID, r, o, d)) = \\
&\quad (r = Rt_A \wedge CnRoute_{LL}(Rt_A, o, d))?; \\
&\qquad \pi t.takeRoad(sID, t, o, L1); takeRoad(sID, Rd_a, L1, d) \mid \\
&\quad (r = Rt_B \wedge CnRoute_{LL}(Rt_B, o, d))?; \\
&\qquad \pi t.takeRoad(sID, t, L2, d) \mid \\
&\quad (r = Rt_C \wedge CnRoute_{LL}(Rt_C, o, d))?; \\
&\qquad \pi t.takeRoad(sID, t, o, L1); takeRoad(sID, Rd_c, L1, d) \mid \\
&\quad (r = Rt_D \wedge CnRoute_{LL}(Rt_D, o, d))?; \\
&\qquad \pi t.takeRoad(sID, t, L3, d) \\
&m^{eg}(deliver(sID)) = unload(sID); getSignature(sID) \\
&m^{eg}(delay(sID)) = delayBW(sID) \mid delayRM(sID) \\
&m^{eg}(Priority(sID)) = Exp1(sID) \vee Exp2(sID) \\
&m^{eg}(Delivered(sID)) = Unloaded(sID) \wedge Signed(sID) \\
&m^{eg}(Delayed(sID)) = DelayedBW(sID) \vee DelayedRM(sID) \\
&m^{eg}(At_{HL}(sID, l)) = At_{LL}(sID, l) \\
&m^{eg}(CnRoute_{HL}(r, o, d)) = CnRoute_{LL}(r, o, d) \\
&m^{eg}(Dest_{HL}(sID, l)) = Dest_{LL}(sID, l)
\end{aligned}
$$

Note that we also need to add the complete specification of $CnRoute_{LL}$ to the initial state axioms of $\mathcal{D}_l^{eg}$. It is straightforward to confirm that $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_h^{eg}$ wrt $m^{eg}$.

**The setp($P$) Construct.** Observe that even if the program associated to each high-level action $m(A_i(\vec{x}))$ is SD, the nondeterministic branch of several high-level actions may not be SD if executions of different high-level actions may share prefixes. E.g., if we have two high-level actions $A$ and $B$, with $m(A) = a_1; a_2$ and $m(B) = a_1; a_3$, then we get $(a_1; a_2) \mid (a_1; a_3)$, which is not SD. After performing the first transition, we are left with either $a_2$ or $a_3$ remaining, and we only have one choice for the next action.

We can address this problem by introducing a new program construct **setp**($P$) that executes a set of programs $P$ nondeterministically without committing to which element of $P$ is being executed unless it has to.[3] The transition semantics for it is as follows:

$$
\begin{aligned}
&Trans(\textbf{setp}(P), s, \delta', s') \equiv \\
&\quad \exists \delta. \exists \delta''. \delta \in P \wedge Trans(\delta, s, \delta'', s') \wedge \\
&\quad \delta' = \textbf{setp}(\{\delta'' \mid \exists \delta. \delta \in P \wedge Trans(\delta, s, \delta'', s')\}) \\
&Final(\textbf{setp}(P), s) \equiv \exists \delta. \delta \in P \wedge Final(\delta, s)
\end{aligned}
$$

Note that **setp**($P$) is always SD. For the example above, **setp**($\{(a_1; a_2) \mid (a_1; a_3)\}$) can make a transition to **setp**($\{a_2, a_3\}$), which can then execute either $a_2$ or $a_3$.

**Monitorable agents.** We assume that the agent only executes low-level action sequences that refine some high-level action sequences, so that the agent is *monitorable*. At the high level, we consider that the agent may do any sequence of executable actions. We define the following high-level programs to capture this:

$$
\begin{aligned}
&\textsc{anyOne} \doteq \mid_{A_i \in \mathcal{A}_h} \pi \vec{x}. A_i(\vec{x}), \text{ do any HL primitive action,} \\
&\textsc{any} \doteq \textsc{anyOne}^*, \text{ i.e., do any sequence of HL actions.}
\end{aligned}
$$

[3] We will assume that the sets of complete refinements of different high-level actions are disjoint (see Assumption 1 below). So once we have finished executing at the low level a sequence $\vec{a}$ that is a refinement of some high-level action, there will be a unique high-level action $\alpha$ that the sequence $\vec{a}$ refines, i.e., such that $Do(m(\alpha), s_l, do(\vec{a}, s_l))$.

This corresponds at the low level to executing refinements of high-level actions/action sequences, which we represent by the following low-level programs:

$$
\begin{aligned}
&\textsc{oneMonit} \doteq \ \textbf{setp}(\{\pi \vec{x}. m(A_i(\vec{x})) \mid A_i \in \mathcal{A}_h\}), \\
&\qquad \text{i.e., do any refinement of any HL primitive action,} \\
&\textsc{monit} \doteq \textsc{oneMonit}^*, \\
&\qquad \text{i.e., do any sequence of refinements of HL actions.}
\end{aligned}
$$

The agent being monitorable means that its possible runs/behaviors are those of MONIT, i.e., the space of possible behaviors of the agent is $\mathcal{CR}_{M_l}(\textsc{monit}, S_0)$. Note that if we have bisimilar models, the converse follows, i.e., any executable high-level action sequence has an executable refinement at the low-level.

**Inverse Mapping.** We want to be able to map a sequence of low-level actions back into a *unique* abstract high-level action sequence it refines. To allow this, as in [7], the following assumption is required:

Assumption 1. *For any distinct ground high-level action terms $\alpha$ and $\alpha'$ it is the case that:*

**(a)** $\mathcal{D}_l \cup C \models \forall s, s'. Do(m(\alpha), s, s') \supset \neg \exists \delta. Trans^*(m(\alpha'), s, \delta, s')$

**(b)** $\mathcal{D}_l \cup C \models \forall s, s'. Do(m(\alpha), s, s') \supset \neg \exists a \exists \delta. Trans^*(m(\alpha), s, \delta, do(a, s'))$

**(c)** $\mathcal{D}_l \cup C \models \forall s, s'. Do(m(\alpha), s, s') \supset s < s'$

Part $(a)$ ensures that different high-level primitive actions have disjoint sets of refinements; $(b)$ ensures that once a refinement of a high-level primitive action is complete, it cannot be extended further; and $(c)$ ensures that a refinement of a high-level primitive action will produce at least one low-level action.

Given this assumption, one can write $m_{M_l}^{-1}(do(\vec{a}, S_0)) = \vec{\alpha}$ to state that $\vec{\alpha}$ is the unique sequence of high-level actions that the low-level action sequence $\vec{a}$ refines; this notation is defined as follows:

$$
\begin{aligned}
&m_{M_l}^{-1}(s) = \vec{\alpha} \doteq M_l \models \exists s'. lp_m(s) = s' \wedge Do(m(\vec{\alpha}), S_0, s') \\
&lp_m(s) = s' \doteq Do(\textsc{monit}, S_0, s') \wedge s' \leq s \wedge \\
&\qquad \forall s''. (s' < s'' \leq s \supset \neg Do(\textsc{monit}, S_0, s''))
\end{aligned}
$$

where, $lp_m(s)$ denotes the largest prefix of $s$ that can be produced by executing a sequence of high-level actions. Note that we extend this notation to apply to any set of action sequences $E_l$ as well, i.e., $m_{M_l}^{-1}(E_l, s_l) = \{\vec{\alpha} \mid \vec{a} \in E_l \text{ and } M_l \models Do(m(\vec{\alpha}), s_l, do(\vec{a}, s_l))\}$.

## 5 HIERARCHICAL AGENT SUPERVISION

To facilitate supervision of agents that have complex behavior, we want to use abstraction. We assume that the supervision specification is expressed as a high-level program, which is quite natural. Given this, we would like to first obtain a supervisor for the high-level agent, and then use it to obtain a supervisor for the low level that ensures that the refinement of the given specification is satisfied.

**Example (cont.)** Going back to our running example, we can represent our supervision specification that says that any shipment that has been ordered, in our case just 123, must eventually be

delivered, and if it is a *Priority* shipment, it should never be delayed, by the following high-level program $\delta^h_{Spec}$:

$$\&_{sID \in ShpOrd}[\pi a.a; (Priority(sID)) \supset \neg Delayed(sID))?]^*; \\ Delivered(sID)?$$

*Ensuring that HL and LL Controllability Match for Atomic Actions.* To be able to use the high-level model to characterize controllable sets of low-level runs of the agent, including the MPS for a given specification, we must first ensure that the formalization of the controllability of individual actions at the high level (in terms of the $A_u$ predicate) accurately reflects the controllability of the actions' implementations at the low level. It is easy to show the following:[4]

Lemma 5.1. *For any $E_s \subseteq \mathcal{A}_h$, $E_s \neq \emptyset$,*

$$\mathcal{D}_h \cup C \models Controllable(\mathbf{set}(E_S), \textsc{anyOne}, s) \equiv \\ \forall a_u.A^u_h(a_u, s) \wedge Poss(a_u, s) \supset Do(\mathbf{set}(E_S), s, do(a_u, s))$$

i.e., at the high level, any set of executable *atomic* actions $E_S$ is controllable wrt the set of all atomic actions the agent may execute in situation $s$ provided that $E_S$ includes all the executable uncontrollable actions in $s$. Indeed, we assume that the supervisor can block any set of controllable high-level actions while leaving the other actions unconstrained.

However, it is easy to construct examples where the low level cannot enforce such supervision specifications. Suppose that we have the high-level actions $\alpha$, $\beta$, and $\gamma$, all of which are controllable and executable in $S_0$, with the following mapping:

$$m(\alpha) = a; u_1 \; m(\beta) = a; u_2 \; m(\gamma) = b.$$

We assume that low-level actions $a$ and $b$ are always controllable and executable and $u_1$ and $u_2$ are always uncontrollable and executable. Suppose that we have a high-level specification which allows the agent to perform only $\alpha$. The high-level MPS could achieve this by not allowing other actions ($\beta$ or $\gamma$) to be executed. However, if we map this specification to the low level, the low-level MPS cannot achieve a similar result, as it needs to allow the execution of $(a; u_1)$ only. However, since $u_2$ is uncontrollable, it may happen and by definition, uncontrollable actions cannot be stopped; thus, there is no way to disable refinements of action $\beta$ without disabling refinements of $\alpha$. In other words, while $\mathbf{setp}(\{m(\alpha), m(\beta)\})$ is controllable, $m(\alpha)$ and $m(\beta)$ when considered individually, are not controllable. The high-level model cannot represent this kind of example by classifying individual actions as controllable or not (using $A^u_h(a_h, s)$).

To enable us to exploit the high-level model of the agent to perform supervision of the low-level agent, we need to ensure that the specification of controllable and uncontrollable actions (i.e., $A^u_h(a, s)$) in the high-level model is consistent with the controllability of the associated programs at the low-level. We do this by assuming that the agent models satisfy the following:

Assumption 2 (Local Controllability). *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$ (wrt $M_h \sim_m M_l$), then*
*(a) for any set of ground high-level actions $E_h$,*
*$M_h \models Controllable(\mathbf{set}(E_h), \textsc{anyOne}, do(\vec{\alpha}, S_0))$*
*if and only if*
*there exists a set of ground low-level action sequences $E_l$ such that*
*$M_l \models Controllable(\mathbf{set}(E_l), \textsc{oneMonit}, do(\vec{a}, S_0))$*
*and $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$;*
*(b) $M_l \models Controllable(\mathbf{set}(\{\epsilon\}), \textsc{monit}, do(\vec{a}, S_0))$*
*if and only if*
*$M_h \models Controllable(\mathbf{set}(\{\epsilon\}), \textsc{any}, do(\vec{\alpha}, S_0))$.*

Intuitively, part (a) ensures that if we have a controllable set of atomic actions (wrt (anyOne in $do(\vec{\alpha}, S_0)$) at the high level, we can always find a set of refinements of exactly these actions that is controllable (wrt (oneMonit in $do(\vec{a}, S_0)$) at the low level; moreover, if we have an uncontrollable set of atomic actions at the high level, there is no set of refinements of exactly these actions that is controllable at the low level, i.e., the set really is uncontrollable. Additionally part (b) ensures that if the supervisor can direct the agent to stop at the low level, i.e., the set of runs $\{\epsilon\}$ is controllable (wrt (monit in $do(\vec{a}, S_0)$), and thus there is no refinement of a high-level action that starts with an uncontrollable low-level action, then the supervisor can also direct the agent to stop at the high level, i.e., $\{\epsilon\}$ is also controllable (wrt (any in $do(\vec{\alpha}, S_0)$) at the high level, and no uncontrollable action is executable there as well, and vice versa (in fact, the latter follows from part (a)).

**Example (cont.)** Suppose that at the low level, the agent has executed $\vec{a} = takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_c, o, d)$ which corresponds to the high-level $\vec{\alpha} = takeRoute(sID, Rt_C, o, d)$. At the high level, the set of all executable actions is $A_h = \{takeRoute(sID, Rt_D, o, d), delay(sID)\}$ and the only controllable subsets are $A_h$, $\{delay(sID)\}$ and $\emptyset$. At the low level, the controllable subsets are $\{takeRoad(sID, Rd_d, o, d), delayBW(sID), delayRM(sID)\}$, $\{delayBW(sID), delayRM(sID)\}$, and $\emptyset$, which correspond to the high-level ones. $\mathbf{set}(\{\epsilon\})$ is not controllable at either levels. So the local controllability assumption is satisfied.

*Hierarchical Controllability of High-Level Specifications.* The local controllability assumption (part (a)) ensures that the controllability of atomic actions at the high level accurately represents the controllability of their refinements. Can we generalize this to show that if we have a controllable set of runs $E_h$ (wrt (any in $do(\vec{\alpha}, S_0)$) at the high level, we can always refine it and obtain a set $E_l$ of runs which are refinements for the runs in $E_h$ and that is controllable (wrt (monit in $do(\vec{a}, S_0)$) at the low level? Indeed we can, as the following result shows:

Theorem 5.2. *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, and Assumptions 1 and 2 (part(a) $\supset$) hold, then for any set of ground high-level action sequences $E_h$ such that $M_h \models Controllable(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$,*
*there exists a set of ground low-level action sequences $E_l$ such that $E_l \subseteq C\mathcal{R}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ and $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$ and $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$.*

We can also show a similar result in the concrete to abstract direction, i.e., if we have a controllable set of refinements of high level action sequences $E_l$, the corresponding set of high-level runs $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))$ must also be controllable at the high level:

Theorem 5.3. *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, and Assumptions 1 and 2 (part(a) $\subset$ and part (b)) hold,*

[4] For proofs of our results, see [8].

*then for any set of ground low-level action sequences $E_l$ such that* $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$
*if* $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$, *then,*
$M_h \models Controllable(\mathbf{set}(m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))), \textsc{any}, do(\vec{\alpha}, S_0))$.

An immediate consequence of the above is that any set of high-level action sequences that is uncontrollable (wrt (ANY in $do(\vec{\alpha}, S_0)$) has no refinement set that is controllable (wrt MONIT in $do(\vec{a}, S_0)$), as if there was such a set, then $E_h$ would have to be controllable by Theorem 5.3.

As we will see, we can use the above results to show that if we have a supervision specification represented by a high-level SD program $\delta^h_{Spec}$, the MPS for the specification at the high level is in fact the abstract version of the MPS for it at the low level. To state this precisely however, we need a way of mapping the high-level supervision specification program $\delta^h_{Spec}$, which is SD, into a low-level program whose runs are the refinements of $\delta^h_{Spec}$.

To support this, we extend the mapping $m$ to a mapping $m_p$ that maps any SD high-level program $\delta^h$ to a SD low-level program that implements it:

$$m_p(\delta^h) \doteq \mathbf{setp}(\{\delta^h[A(\vec{t})/\mathbf{atomic}(m(A(\vec{t}))) \text{ for all } A \in \mathcal{A}, \text{ and } F(\vec{t})/m(F(\vec{t})) \text{ for all } F \in \mathcal{F}]\}).$$

Note that when we replace a high-level action $A(\vec{t})$ by the low-level program implementing it, $m(A(\vec{t}))$, we enclose the latter in the **atomic**() construct to prevent it from being interleaved with refinements of other high-level actions, as we want any low-level execution of the agent to be a sequence of refinements of high-level actions. We also use the **setp**() construct to avoid committing to a particular high-level action that is being refined until we have to.

We can then use $m_p$ to map an arbitrary supervision specification represented by a high-level SD program $\delta^h_{Spec}$ to the SD low-level program that implements it $m_p(\delta^h_{Spec})$.

**Example (cont.)** Applying $m_p$ to the high-level specification $\delta^h_{Spec}$ given earlier yields the following low-level specification:

$$m_p(\delta^h_{Spec}) = \delta^l_{Spec} = \\ \&_{sID \in ShpOrd}[\pi a.a; ((Exp1(sID) \vee Exp2(sID)) \supset \neg(DelayedBD(sID) \vee DelayedRM(sID)))?]^*; \\ (Unloaded(sID) \wedge Signed(sID))?$$

Now we are ready to state our result: the high-level MPS for the supervision specification represented by a high-level SD program $\delta^h_{Spec}$ is the abstract version of the MPS for the mapped specification at the low level, i.e., formally:

THEOREM 5.4. *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, and Assumptions 1 and 2 hold, then for any supervision specification represented by a high-level situation-determined program $\delta^h_{Spec}$,*

$$m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps(\textsc{monit}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0)), do(\vec{a}, S_0)) = \\ \mathcal{CR}_{M_h}(mps(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)).$$

**Example (cont.)** The complete runs of the high-level MPS and low-level MPS are as follows:

$\mathcal{CR}_{M_h}(mps(\textsc{any}, \delta^h_{Spec}, S_0), S_0) =$
$\{takeRoute(sID, Rt_A, o, d); takeRoute(sID, Rt_B, o, d); deliver(sID)\}$

$\mathcal{CR}_{M_l}(mps(\textsc{monit}, m_p(\delta^h_{Spec}), S_0), S_0) =$
$\{[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d); takeRoad(sID, Rd_{b1}, o, d); unload(sID); getSignature(sID)],$
$[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d); takeRoad(sID, Rd_{b2}, o, d); unload(sID); getSignature(sID)]\}$

It is easy to confirm that the result of Theorem 5.4 holds.

## 6 HIERARCHICALLY SYNTHESIZED MPS

Let's assume that we have precomputed the high-level MPS for some high-level specification in some high-level situation, which for convenience we equivalently represent as a set of high-level action sequences $E_h^{mps}$. We can define a low-level program $mps_i(E_h^{mps})$ that refines this high-level MPS $E_h^{mps}$ into the corresponding low-level MPS (note that *now* represents the current situation):

$$mps_i(E_h^{mps}) = \epsilon \in E_h^{mps}? \mid \\ (mps(\textsc{oneMonit}, m_p(firsts(E_h^{mps})), now); \\ mps_i(rests(E_h^{mps}, last(m_{M_l}^{-1}(now)))))),$$

where
$last(\vec{\gamma}) = \beta$ if $\vec{\gamma} = \vec{\alpha}'\beta$ and undefined if $\vec{\gamma} = \epsilon$,
$firsts(E) = \{\alpha' \mid \alpha'\vec{\gamma} \in E \text{ for some } \vec{\gamma}\}$, and
$rests(E, \beta) = \{\vec{\gamma} \mid \beta\vec{\gamma} \in E\}$.

We can show that the resulting *hierarchically synthesized* MPS, $mps_i(E_h^{mps})$, is *correct* in that it has exactly the same set of complete runs as that of the low-level MPS $mps(\textsc{monit}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$ obtained by mapping the supervision specification $\delta^h_{Spec}$ to the low level:

THEOREM 6.1. *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, and Assumptions 1 and 2 hold, then for any supervision specification represented by a high-level situation-determined program $\delta^h_{Spec}$*

$$\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) = \\ \mathcal{CR}_{M_l}(mps(\textsc{monit}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0))$$
*where* $E_h^{mps} = \mathcal{CR}_{M_h}(mps(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$.

The hierarchically synthesized MPS $mps_i(E_h^{mps})$ will generally be much easier to compute than the low-level MPS $mps(\textsc{monit}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$. To get the latter, one has to search the whole space of all refinements of all high-level action sequences. To get the former, one only needs to repeatedly search for the local MPS of the set of refinements of high-level atomic actions that are allowed by the high-level MPS at each step; the search horizon is much shorter, a single high-level action. One does need to precompute the high-level MPS $mps(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0))$, but the search space for this would typically be much smaller than for the low-level MPS.

One may also compute $mps_i(E_h^{mps})$ incrementally. The fact that we have the high-level MPS as a guide ensures that we can do this without losing maximal permissiveness. Note that $mps_i(E_h^{mps})$ is a sequence of $\mathbf{set}(E)$, so it is always SD, like the low-level MPS, thus ensuring that they can always perform the same transitions.

**Example (cont.)** Initially, $mps(\text{MONIT}, m_p(\delta^h_{Spec}), S_0)$ needs to consider both of the following action sequences:

$$[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d)]$$
$$[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_c, o, d)]$$

After the latter, the uncontrollable actions $delayRM$ or $delayBW$ may happen next, which would violate the specification given that shipment 123 is of type $Exp2$. So it will only include the former as a prefix in the resulting MPS. $mps_i(E_h^{mps})$ however, only needs to consider refinements of $takeRoute(sID, Rt_A, o, d)$, i.e., $[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d)]$. The high-level MPS $mps(\text{ANY}, \delta^h_{Spec}, S_0)$ has already decided that taking route $C$ should not be allowed, as it may be followed by the uncontrollable action $delay$, which is ruled out by the specification since 123 is of type $Priority$.

Observe that, so far, we have assumed that we have complete information about the situation in which the agent runs in both the high-level and low-level models. But this is not essential. If the high-level MPS for the given supervision specification is the same in all models of the high-level action theory (i.e., it is similar to a conformant plan) and we have a sound abstraction, then we can still use this high-level MPS to obtain a correct hierarchically synthesized MPS for each low-level model as shown above. Of course, if we have incomplete information at the low level too, then there is no guarantee that the resulting low-level MPS will be the same for every model of the low-level action theory. However, one typically has more information at the low level than at the high level, so this case is not that unusual. More generally, an agent with incomplete information may also acquire new information online, as it executes. In this case, a more complex notion of online supervision/MPS is required [6]. Extending our hierarchical approach to this case is left for future work.

## 7 DISCUSSION

Our approach is inspired by the hierarchical supervisory control of discrete event systems [33, 34, 37], but the foundations of our work is different: the framework is based on a rich first-order logic language; we use a notion of bisimulation to relate the models of the high-level and low-level theories; in addition to actions (which abstract over programs), our high-level theory includes fluents (which abstract over formulas); and through preconditions for actions, we are able to enforce local constraints on the low-level agent.

Aucher [5] reformulates the results of supervisory control theory in terms of model checking problems in an epistemic temporal logic. van der Hoek et al. [31] formalize how the abilities of coalitions of agents are affected by the transfer of control over variables. Alechina et al. [1] regulate multi-agent systems using norms. Unlike our work, these approaches are not based on first order logic. Gabaldon [20] incorporates norms as preconditions of actions in Golog [22]. The above approaches do not consider abstraction.

Grossi and Dignum [21] use a KD45 multi-modal logic corresponding to a propositional logic of contexts to model norms at different levels of abstraction. Also, Salceda and Dignum [32] propose a method to refine abstract norms specified in institutional regulations to concrete norms and eventually into rules and procedures represented in PDL [25] such that agents in the organization can be rewarded/punished based on these norms. These approaches use propositional logics. There is also related work in the area of model checking and synthesis of hierarchical systems [2–4, 9].

In AI, behavior composition [17, 36] involves synthesizing a controller that realizes a virtual target behavior by coordinating the execution of set of available behavior modules. A notion of *controller generator* (i.e., an implicit representation of all controllers) is also studied. Felli et al. [18] relate the controller generator to a notion of MPS. These approaches model behaviors as finite state transition systems. Sardina et al. [30] synthesize a controller that orchestrates the concurrent execution of a library of available ConGolog programs to realize a virtual target program, but the controller is not maximally permissive. The above approaches do not consider abstraction.

In planning, several notions of abstraction, including *precondition-elimination abstraction*, *Hierarchical Task Networks* (HTNs) and *macro operators* have been studied [28]. HTNs in ConGolog [19] and *complex actions* specified as Golog programs [24] have also been investigated. While these approaches focus on improving the efficiency of planning, our work provides a generic framework for customizing the agent behavior. Moreover, the former uses a single BAT, and the latter compile the abstracted actions into a new BAT that contains both the original and abstracted actions. Also, the latter only deals with deterministic complex actions and does not provide abstraction for fluents.

In this paper, we developed an account for hierarchical supervision where given a high-level MPS based on an abstract specification, we synthesize a MPS for the low-level agent based on the refined specification. For simplicity, we focused on a single layer of abstraction, but the framework supports extending the hierarchy to more levels. Our approach can be extended to use ConGolog programs (in addition to the action theory) to specify the possible behaviors of the agent at both the high and low level; one way to do this is to "compile" the program into the BAT $\mathcal{D}$ to get a new BAT $\mathcal{D}'$ whose executable situations are exactly those that can be reached by executing the program, as in [14]. In future work, we will explore how "compatible" low-level specifications on the concrete agent behavior can also be incorporated into the low-level MPS. Moreover, we will investigate an account of hierarchical supervision for agents that execute online and can acquire new information (e.g., through sensing) as they operate. Another direction for future research is investigating how the local controllability condition can be verified.

The framework developed in this paper is very general and handles arbitrary first-order representations of the dynamic system's states. In such a general setting not much can be said about the computational aspects. However note that we can get an effective setting from the computational point of view if we restrict, for example, the high level to be propositional. In this way we get a finite

state abstract system on which doing supervision becomes effective, (under the assumption that we are able to compute refinements of atomic actions). Similar results can be obtained for first-order bounded action theories [13]. We leave these questions for future work.

## ACKNOWLEDGMENTS

We acknowledge the support of Sapienza Ateneo Project ŞImmersive Cognitive EnvironmentsŤ and the National Science and Engineering Research Council of Canada.

# REFERENCES


[1] Natasha Alechina, Nils Bulling, Mehdi Dastani, and Brian Logan. 2015. Practical Run-Time Norm Enforcement with Bounded Lookahead. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*. ACM, 443–451.

[2] Rajeev Alur and Mihalis Yannakakis. 2001. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 273–303.

[3] Benjamin Aminof, Orna Kupferman, and Aniello Murano. 2012. Improved model checking of hierarchical systems. *Information and Computation* 210 (2012), 68–86.

[4] Benjamin Aminof, Fabio Mogavero, and Aniello Murano. 2014. Synthesis of hierarchical systems. *Science of Computer Programming* 83 (2014), 56–79.

[5] Guillaume Aucher. 2014. Supervisory Control Theory in Epistemic Temporal Logic. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*. Springer.

[6] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. 2016. Online Agent Supervision in the Situation Calculus. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. IJCAI/AAAI Press, 922–928.

[7] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. 2017. Abstraction in Situation Calculus Action Theories. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI Press, 1048–1055.

[8] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. 2018. *Hierarchical Agent Supervision - Extended Version.* Technical Report EECS-2018-01. York University.

[9] Laura Bozzelli, Aniello Murano, Giuseppe Perelli, and Loredana Sorrentino. 2017. Hierarchical Cost-Parity Games. In *24th International Symposium on Temporal Representation and Reasoning, TIME*. 6:1–6:17.

[10] C. G. Cassandras and S. Lafortune. 2008. *Introduction to Discrete Event Systems* (second ed.). Springer.

[11] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. 2000. ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence* 1, 1–2 (2000), 109–169.

[12] Giuseppe De Giacomo, Yves Lespérance, and Christian J. Muise. 2012. On supervising agents in situation-determined ConGolog. In *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012*. IFAAMAS, 1031–1038.

[13] G. De Giacomo, Y. Lespérance, and F. Patrizi. 2016. Bounded situation calculus action theories. *Artificial Intelligence* 237 (2016), 172–203.

[14] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Sebastian Sardiña. 2016. Verifying ConGolog Programs on Bounded Situation Calculus Theories. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press, 950–9568.

[15] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. 2010. Situation Calculus Based Programs for Representing and Reasoning about Game Structures. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 12th International Conference*.

[16] Giuseppe De Giacomo, Hector J. Levesque, and Yves Lespérance. 2004. Trans and Final for Mutual Exclusive Blocks. (2004). Unpublished Note.

[17] Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardina. 2013. Automatic behavior composition synthesis. *Artificial Intelligence* 196 (2013), 106–142.

[18] Paolo Felli, Nitin Yadav, and Sebastian Sardiña. 2017. Supervisory Control for Behavior Composition. *IEEE Trans. Automat. Control* 62, 2 (2017), 986–991.

[19] Alfredo Gabaldon. 2002. Programming hierarchical task networks in the situation calculus. In *AIPSŠ02 Workshop on On-line Planning and Scheduling*.

[20] Alfredo Gabaldon. 2011. Making Golog Norm Compliant. In *Proceedings of the 12th International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 275–292.

[21] Davide Grossi and Frank Dignum. 2004. From Abstract to Concrete Norms in Agent Institutions. In *Proceedings of the 3rd International Workshop on Formal Approaches to Agent-Based Systems*, Vol. 3228. Springer, 12–29.

[22] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programing* 31, 1-3 (1997), 59–83.

[23] J. McCarthy and P. J. Hayes. 1969. Some Philosophical Problems From the Standpoint of Artificial Intelligence. *Machine Intelligence* 4 (1969), 463–502.

[24] Sheila A. McIlraith and Ronald Fadel. 2002. Planning with Complex Actions. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*. 356–364.

[25] John-Jules Ch. Meyer. 1988. A different approach to deontic logic: deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic* 29, 1 (1988), 109–136.

[26] Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. William Kaufmann, 481–489.

[27] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[28] Dana Nau, Malik Ghallab, and Paolo Traverso (Eds.). 2004. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc.

[29] Ray Reiter. 2001. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

[30] Sebastian Sardiña and Giuseppe De Giacomo. 2009. Composition of ConGolog Programs. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. 904–910.

[31] Wiebe van der Hoek, Dirk Walther, and Michael Wooldridge. 2014. Reasoning About the Transfer of Control. *CoRR* abs/1401.3825 (2014).

[32] Javier Vázquez-Salceda and Frank Dignum. 2003. Modelling Electronic Organizations. In *Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems (Lecture Notes in Computer Science)*, Vol. 2691. Springer, 584–593.

[33] KC Wong and WM Wonham. 1996. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems* 6, 3 (1996), 783–800.

[34] WM Wonham. 2017. *Supervisory Control of Discrete-Event Systems* (2017 ed.). University of Toronto.

[35] WM Wonham and PJ Ramadge. 1987. On the supremal controllable sub-language of a given language. *SIAM J Contr Optim* 25, 3 (1987), 637–659.

[36] Nitin Yadav, Paolo Felli, Giuseppe De Giacomo, and Sebastian Sardina. 2013. Supremal Realizability of Behaviors with Uncontrollable Exogenous Events. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 1176–1182.

[37] H Zhong and WM Wonham. 1990. On consistency of hierarchical supervision in discrete-event systems. *IEEE Trans. Automat. Control* 35, 10 (1990), 1125–1134.