ONLINE AND HIERARCHICAL AGENT SUPERVISION

BITA BANIHASHEMI

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
DECEMBER 2017

ONLINE AND HIERARCHICAL AGENT SUPERVISION

by **Bita Banihashemi**

a dissertation submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

© 2018

ONLINE AND HIERARCHICAL AGENT SUPERVISION

by **Bita Banihashemi**

Examination Committee Members:

1. Zbigniew Stachniak (Chair)
2. Yves Lespérance (Supervisor)
3. Sotirios Liaskos (Internal Member)
4. Marin Litoiu (Committee Member)
5. Jonathan Ostroff (Committee Member)
6. Evgenia Ternovska (External Examiner)

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. This framework is based on the situation calculus and a variant of the ConGolog agent programming language. In this dissertation, we focus on two of the open problems with the original account of agent supervision. The first open problem is supervising an agent that may acquire new knowledge about her environment during an online execution, for example, by sensing. The second open problem concerns the supervision of agents that operate in complex domains and have complex behavior. Such agents typically need to represent and reason about a large amount of knowledge. One approach to cope with this challenge is to use abstraction, which involves developing an abstract/high-level model of the agent behavior that suppresses less important details. Hence, we first investigate abstracting an agent's behavior in offline executions, and formalize a notion of sound and/or complete abstractions. Sound abstractions can be used to perform several forms of reasoning about action, such as planning, agent monitoring, and generating high-level explanations of low-level/concrete agent behavior. Moreover, we investigate abstraction of agent's behavior in online executions, and discuss its relation to hierarchical contingent planning. We then use our results on offline agent abstraction to formalize hierarchical agent supervision: in a first step, we only consider the high-level model and obtain the maximally permissive supervisor to customize the abstract agent behavior; then in a second step, we obtain a low-level supervisor by refining the high-level supervisor's actions locally. We show that this process can be done incrementally, without precomputing the local refinements.

To My Parents

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Yves Lespérance, for his guidance, encouragement and patience throughout the development of this dissertation. I truly appreciate his contributions of time and ideas that made this dissertation possible. I am also deeply grateful to our collaborator, Professor Giuseppe De Giacomo, for his insightful comments and suggestions. I have been privileged to work with both of them and be inspired by their broad knowledge and deep way of thinking.

A special thanks to the members of my supervisory committee, Professor Marin Litoiu and Professor Jonathan Ostroff for their suggestions and constructive comments. I am also thankful to my external examiner Professor Evgenia Ternovska, for coming to Toronto to attend my oral exam, and for her thorough review of the dissertation and feedback.

I would also like to thank Professor Gerhard Lakemeyer and Professor Sheila McIlraith for their feedback and suggestions on online agent supervision during the earlier stages of this work.

Finally, I am forever indebted to my family for their love, encouragement, and support.

Table of Contents

List of Figures

Abbreviations

Acronyms

Acronym	Description	Page Defined
SCDES	Supervisory Control of Discrete Event Systems	35
BAT	Basic Action Theory	47
SSA	Successor State Axiom	48
SD	Situation-Determined	55
MPS	Maximally Permissive Supervisor	57
HL	High Level	78
LL	Low Level	78

Notations

Notation	Description	Page Introduced
$\vec{a}$	Sequence of actions	46
$SituationDetermined(\delta, s)$	δ is situation determined in s	55
$\mathcal{RR}_{\mathit{offl}}(\delta, s)$	Offline partial runs of δ in situation s	56
$\mathcal{GR}_{\mathit{offl}}(\delta, s)$	Offline good runs of δ in situation s	56
$\mathcal{CR}_{\mathit{offl}}(\delta, s)$	Offline complete runs of δ in situation s	56
$Controllable(\delta_s, \delta_i, s)$	δ_s is controllable wrt δ_i in situation s	57
$mps_{\mathit{offl}}(\delta_i, \delta_s, s)$	Offline maximally permissive supervisor of agent behavior δ_i for specification δ_s in situation s	57
$\mathbf{set}(E)$	infinitary nondeterministic branch over the set of action sequences E	57
$\&_{A_u}$	Offline supervision construct	58

Notations Cont.

Notation	Description	Page Introduced
$\sigma = \langle \mathcal{D}, \delta_i \rangle$	Agent σ with initial program δ_i and BAT $\mathcal{D}$	64
$\langle \delta, \vec{a} \rangle$	Agent configuration with remaining program δ at current history $\vec{a}$	64
$\langle \delta, \vec{a} \rangle \rightarrow_{A(\vec{n})} \langle \delta', \vec{a}A(\vec{n}) \rangle$	Online transition relation (involving action $A(\vec{n})$)	65
$c^{\checkmark}$	Configuration c is final	65
$\mathcal{RR}(\sigma)$	Online partial runs of σ	66
$\mathcal{GR}(\sigma)$	Online good runs of σ	66
$\mathcal{CR}(\sigma)$	Online complete runs of σ	66
$mps_{onl}(\delta_s, \sigma)$	Online maximally permissive supervisor of agent σ for specification δ_s	68
$\&^{onl}_{A_u}$	Online supervision construct	69
$\Sigma^{w}_{onl}(\delta_s, \delta_i)$	Weak online search construct over specification δ_s and agent behavior δ_i	71
$s_h \sim^{M_h, M_l}_{m} s_l$	Situation s_h in model M_h is isomorphic to situation s_l in M_l wrt mapping m	82
$M_h \sim_m M_l$	Model M_h is bisimilar to model M_l wrt mapping m	82
$m^{-1}_{M_l}(S)$	Inverse mapping m^{-1} for situation S in model M_l	90
$m^{-1}_{M_l}(E_l, s_l)$	Inverse mapping m^{-1} for set of action sequences E_l in situation s_l in model M_l	94
$\mathcal{CR}_M(\delta, s)$	Offline complete runs of agent δ in situation s in model M	95
$\mathbf{setp}(P)$	delayed commitment nondeterministic branch over the set of programs P	95
$m_p(\delta_h)$	maps any situation-determined high-level program δ_h to a situation-determined low-level program that implements it	102
$mps_i(E^{mps}_h)$	Low-level program that refines high-level MPS E^{mps}_h into the corresponding low-level MPS	104
$AbleBy(\delta, \vec{a}, \gamma)$	Online situation-determined program (i.e., task) δ can successfully be executed in situation $do(\vec{a}, s)$ by following strategy γ	121

1 Introduction

Knowledge representation and reasoning is a subfield of Artificial Intelligence (AI) that focuses on how knowledge can be represented symbolically and manipulated in an automated way by reasoning programs. More informally, it is a part of AI that is concerned with cognition, and how thinking contributes to intelligent behavior [21]. Reasoning about action and change (RAC) is a prominent area of research within this field that focuses on representation and reasoning about actions of intelligent agents (as well as other actors in the environment) operating in dynamical domains.

Such agents typically have knowledge of when a certain action is executable, and how it affects the world. Moreover, they often have (incomplete) beliefs about the state of the world and its dynamics. These agents may also be able to perform sensing during execution to acquire additional knowledge about the world. Such properties enable agents to reason about what action to perform in each situation to ensure they can execute their task and/or achieve their goals.

In many settings, an agent's behavior needs to be restricted to conform to a set of specifications. For instance, the activities of agents in an organization have to adhere to business rules and privacy/security protocols. Similarly, a mobile robot has to conform to safety specifications and avoid causing injuries to others. One form of restricting an agent's behavior is *customization*, where a generic agent behavior for performing a task or achieving a goal, typically involving several alternative courses of actions, is refined to satisfy a client's constraints or preferences.

As intelligent agents are capable of reasoning about actions and exhibiting autonomous behavior, a key challenge in such settings is ensuring *conformance* to specifications while preserving the agent's *autonomy*. In the area of reasoning about action and change, motivated by this challenge and inspired by the Supervisory Control of Discrete Event Systems (SCDES) [166, 24], De Giacomo, Lespérance and Muise (DLM) [35] proposed *agent supervision*, as a form of control/customization of an agent's behavior. This framework is based on the situation calculus [110, 132], a predicate logic language designed for representing and reasoning about dynamically changing worlds and a variant of the ConGolog agent programming language [33]. In this variant, all programs are assumed to be *situation-determined* (SD), meaning that a (partial) execution of the program is *uniquely* determined by the sequence of actions it has produced. DLM represent the agent's possible behaviors as a (non-deterministic) SD ConGolog program. Another SD ConGolog program represents the supervision specification, i.e., which behaviors are acceptable/desirable. If it is possible to control all of the agent's actions, then it is straightforward to specify the result of supervision as the intersection of the execution of the agent and the supervision specification programs. 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. In this context, *controllability* of a supervision specification with respect to the agent program in a situation is informally defined as follows: given any sequence of actions that is a prefix of a complete run of both the supervision specification and the agent program in a given situation, if an uncontrollable action is executable by the agent after this sequence of actions, then the uncontrollable action must also be executable in the specification after the sequence of actions. Given a specification, a *supervisor* is a component that observes the run of the agent so far, and disallows some of the controllable actions in order to satisfy the specification. A supervisor is *maximally permissive* if it disallows actions that cause the agent to violate the specification, but leaves the agent as much freedom as possible to choose among the remaining actions. We will provide formal definitions of situation-determined programs, controllability of a specification with respect to agent behavior, and maximally permissive supervisors in Chapter 3. Note that due to its first-order logic foundations, the agent supervision framework can handle infinite states, and thus generalizes most approaches to SCDES which are based on finite state settings.

Note that since one could implement a new agent that satisfies the specification by design, it may seem that having a supervisor only adds to the complexity of the system. Here however, we aim to customize the generic behavior of an *existing* agent, and we are not focused on designing an agent from scratch that satisfies some specification. In either case, one of the main advantages of this approach is that it promotes system evolvability: as user requirements change, agents are relocated to new environments and new protocols come into place, the agent behavior does not need to be re-defined.

1.1 Motivation

In this dissertation, we aim to generalize the original agent supervision framework proposed by DLM to deal with the following two open problems:

1. **Supervising Online Agents.** The original DLM account of agent supervision assumes that the agent does not acquire new information while executing and does not perform sensing. In other words, it considers an agent's *offline* executions [39]. Typically, agents work in settings where they acquire new knowledge through sensing and exogenous actions. Consider for instance a travel planner agent that needs to book a seat on a certain flight. Only after querying the airline web service offering that flight will the agent know if there are seats available on the flight.

2. **Supervising Complex Agents.** In more complex domains, agents need to represent and reason about a large amount of knowledge about their environment and have complex behaviors. Due to complexity of the behavior logic, designing and enforcing specifications for customization of agent's behavior can be a difficult task. Consider for example a logistics planner agent that needs to plan the shipment of items between various locations. Such an agent has to choose among a network of roads while considering a number of factors such as temporary road closures, municipal rules with respect to usage of roads by heavy vehicles, bad weather, service level agreements regarding priority shipments, etc., that could all affect choices of available roads.

Customization and control of systems is appealing to various research communities; examples include specialization of scientific workflows [73], customization of the behavioral aspect of software systems [106], automation of collaboration between two different business processes [104], increasing the accuracy of programs generated through using the programming by instruction technique [60] and configuration of products in software product lines [5, 154].

One may view *behavior* (i.e., the logic of any artifact operating in an environment) or *web service* composition as a type of process customization, as it involves selecting some available behaviors/web services and controlling their interaction to produce the desired behavior or service. Some major approaches to automated service composition, for instance [113], treat it essentially as the problem of customizing a generic process.

Similarly, in the *Internet of Things* (IoT) [6] field, advanced functionalities (e.g., home surveillance) can be achieved through the cooperation of many simple yet heterogeneous robotic devices pervasively embedded in the environment (e.g., video cameras or mobile phones). The IoT has wide area of applications, for example, industrial automation, elderly assistance, intelligent energy management and traffic management.

1.2 Our Approach and Contributions

Our framework is based on the situation calculus [110, 132], and a variant of the ConGolog agent programming language [33] that assumes all programs are situation-determined. Moreover, through several examples, we motivate our work and demonstrate how our framework can be applied in different domains.

1.2.1 Online Agent Supervision

To accommodate agents that acquire knowledge during executions, we consider the agent's *online* executions [39, 140], where she must make decisions based on what she knows, and her knowledge may be updated as she executes the program. The agent's knowledge base is represented by a situation calculus basic action theory. Similar to DLM, we represent the agent's possible behaviors as well as the supervision specification (i.e., which behaviors are acceptable/desirable) as situation-determined ConGolog programs. We first define a meta-theoretic[1] *online transition relation* between agent configurations. Then, to ensure that for any sequence of actions that the agent can perform in an online execution, there is a *unique* resulting agent configuration, i.e., agent belief state and remaining program, we formalize the notion of *online situation-determined* agents.

Next, we provide a formalization of the *online maximally permissive supervisor* (online MPS) and show its existence and uniqueness. The online supervisor can depend on the information that the agent acquires as it executes; it can also deal with the fact that an action's controllability may not be known at the outset. The online supervisor ensures that the agent behavior satisfies the supervision specification no matter how the world turns out to be.

Moreover, we meta-theoretically define a program construct, the *supervision operator* for online supervised execution that given the agent and specification, executes them to obtain only runs allowed by the maximally permissive supervisor, showing its soundness and completeness. To ensure that the agent under the supervision operator construct considers only runs that can be successfully completed (i.e., ensure non-blockingness), we also define a new *lookahead search* construct.

1.2.2 Agent Abstraction and Hierarchical Agent Supervision

To supervise an agent with complex behaviors, the approach we use is *abstraction*. In essence, it involves developing an abstract model of the agent/domain that suppresses less important details. Abstraction is an interesting topic in its own right and has been investigated in various areas of AI and Computer Science to improve efficiency in planning (e.g., [112]), facilitate verification (e.g., [118]), enable automated reasoning (e.g., [74]), capture the social practices of multi-agent systems (e.g., [50]), facilitate model checking (e.g., [28]) and data integration (e.g., [97]). In first-order settings, while some approaches (e.g., [74, 122]) have formalized a general framework for abstraction of "static" logical theories,[2] there does not seem to be any work that focuses on a general framework for abstraction of dynamic domains.

Abstraction can provide a number of benefits; for example, the abstract model typically allows us to reason more easily about the agent's possible behaviors and to provide high-level explanations of the agent's behavior. Moreover, to efficiently solve a complex reasoning problem, one may first try to find a solution in the abstract model, and then use this abstract solution as a template to guide the search for a solution in the concrete model. Systems developed using abstractions are typically more robust to change, as adjustments

[1] We model the agent's acquiring new information during an online execution by using updated theories, that include the initial theory together with the new knowledge acquired during different stages of execution. Thus, we define an online transition relation that holds over agent configurations that include such updated theories as mathematical objects at the metalevel.

[2] A static theory describes a single non-changing state of affairs; this is in contrast to dynamic theories, such as situation calculus basic action theories, where due to execution of actions, value of fluents may change.

to more detailed levels may leave the abstract levels unchanged. Also, it may be possible to adapt solutions for the abstract level to a large spectrum of the concrete problems.

Hence, we divide the work addressing this open problem in terms of *abstraction of agent behavior* and *hierarchical agent supervision*, where we first obtain a maximally permissive supervisor based on a abstract/high-level model of the agent and then utilize it to obtain a maximally permissive supervisor for the concrete/low-level model of the agent.

1.2.2.1 Abstraction of Offline Agent Behavior

We assume we have a high-level/abstract action theory and a low-level/concrete action theory both representing the agent behavior at different levels of detail and a refinement mapping between the high-level and the low-level action theories. The mapping associates each high-level primitive action to a (possibly nondeterministic) ConGolog program defined over the low-level action theory that "implements it". Moreover, it maps each high-level fluent to a state formula in the low-level language that "characterizes the concrete conditions" under which it holds.

In this setting, we define a notion of a high-level theory being a *sound abstraction* of a low-level theory under a given refinement mapping. The formalization involves the existence of a suitable bisimulation relation [115, 117] between models of the low-level and high-level theories. 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 executability of refinement of a sequence of high-level actions that achieve a refinement of a certain condition is satisfiable at the low level, then the executability of the sequence of high-level actions that achieve a certain condition is satisfiable at the high level. Thus, sound abstractions can be used to perform effectively several forms of reasoning about action, such as planning, agent monitoring, and generating high-level explanations of low-level behavior. We also provide a proof theoretic characterization that gives us the basis for automatically verifying that we have a sound abstraction. In addition, we define a dual notion of *complete abstraction*.

We also identify a set of constraints that ensure that for any low-level action sequence, there is a unique high-level action sequence that it refines. This is useful for providing high-level explanations of behavior and agent monitoring.

1.2.2.2 Hierarchical Agent Supervision in Offline Executions

To facilitate agent supervision for complex agents, and inspired by the hierarchical supervisory control of discrete event systems [166], we build on the results of Section 1.2.2.1 and formalize a framework for *hierar-*

chical agent supervision in the context of offline executions, where the agent does not acquire new knowledge during a run. We assume we have a low-level/concrete basic action theory and a high-level/abstract basic action theory, both representing the agent behavior at different levels of detail and a refinement mapping between the two. We further assume that the high-level basic action theory is a sound abstraction of the low-level basic action theory relative to the refinement mapping. Moreover, the low-level behavior of the agent 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 situation-determined ConGolog program, which specifies the behaviors that are acceptable/desirable. Our task is to synthesize a maximally permissive supervisor for the low-level agent behavior and specification (which we can translate into a low-level program).

To facilitate this task, we 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. We define a new program construct that executes a set of programs P non-deterministically without committing to which element of P being executed unless it has to. With the help of this construct and the constraints identified above, we show that the low-level MPS for the mapped specification is a refinement of the high-level MPS for the specification. Moreover, we show that we can obtain the low-level MPS incrementally by using the high-level MPS as a guide and refining its actions locally while remaining maximally permissive. The resulting hierarchically synthesized MPS has exactly the same runs as that of the low-level MPS obtained by mapping the supervision specification to the low-level. Our approach is inspired by the hierarchical supervisory control of discrete event systems [166], 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.

1.2.2.3 Abstraction of Online Agent Behavior

We build on the results of abstraction of offline agent behavior to formalize a general abstraction framework for an agent that executes online, i.e., can acquire new information as it executes. Similar to Section 1.2.2.1, we assume that we have a high-level/abstract action theory and a low-level/concrete action theory and a refinement mapping between the two.

To formalize a notion of sound abstraction in online executions, we first identify a sufficient condition that ensures a high-level basic action theory remains a sound abstraction of a low-level basic action theory with respect to a refinement mapping as the agent acquires new knowledge. We then extend the results on basic properties of sound abstractions in offline executions to online executions.

To be able to reliably execute a task/achieve a goal in online executions, an agent must have a strategy. We formalize a model of contingent planning over the agent's online executions that ensures it has a strategy to only perform actions that can be extended to a successfully terminating execution of the program, no matter how the environment behaves. Contingent planning involves synthesizing such a strategy. We then show that under some reasonable conditions, if we have sound abstraction and the agent has a strategy for accomplishing a task or achieving a goal at the high level, it is possible to refine it into a low-level strategy piecewise, and the resulting low-level strategy is guaranteed to achieve the refinement of the goal.

1.2.3 Contributions

The main contributions of this dissertation are as follows:

1. **Online Agent Supervision.** The results of this contribution have been presented in [14, 13, 12].

    - We define a notion of *online situation-determined agent* which ensures that for any sequence of actions that the agent can perform online, the resulting agent configuration (i.e., belief state and remaining program) is unique (Section 4.2.2).
    - We provide a formalization of the *online maximally permissive supervisor* and show its existence and uniqueness (Section 4.3.2).
    - Moreover, we meta-theoretically define a *program construct* (i.e., supervision operator) for online supervised execution that given the agent and specification, executes them to obtain only runs allowed by the online maximally permissive supervisor, showing its soundness and completeness (Section 4.3.3). To ensure the agent under the supervision operator construct considers only runs that can be successfully completed (i.e., ensure non-blockingness), we also define a new *lookahead search construct* (Section 4.3.4).

2. **Abstraction of Offline Agent Behavior.** The results of this contribution have been presented in [15, 11].

    - We formalize a notion of a high-level basic action theory being a *sound abstraction* of a low-level basic action theory under a given refinement mapping. This notion is based on a suitable notion of bisimulation between models of the high-level and low-level theories. We also provide a proof theoretic characterization that gives us the basis for automatically verifying that we have a sound abstraction (Section 5.3). In addition, we define a dual notion of *complete abstraction* (Section 5.4).

- We also identify a set of constraints that ensure that for any low-level action sequence, there is a unique high-level action sequence that it refines. This is useful for providing high-level explanations of behavior and monitoring (Section 5.5).

3. **Hierarchical Agent Supervision.** The results of this contribution have been submitted [16].

   - We 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 (Section 6.3).
   - We define a new program construct that executes a set of programs P non-deterministically without committing to which element of P being executed unless it has to (Section 6.1.2). With the help of this construct and the constraints identified above we show that the low-level MPS for the mapped specification is a refinement of the high-level MPS for the specification (Section 6.3).
   - Moreover, we show that we can obtain the low-level MPS incrementally by using the high-level MPS as a guide and refining its actions locally while remaining maximally permissive. We then show that the resulting hierarchically synthesized MPS has exactly the same runs as that of the low-level MPS obtained by mapping the supervision specification to the low-level (Section 6.4).

4. **Abstraction of Online Agent Behavior.**

   - We first identify a sufficient condition that ensures a high-level basic action theory remains a sound abstraction of a low-level basic action theory with respect to a refinement mapping as the agent acquires new knowledge (Section 7.2).
   - We formalize a model of contingent planning over agent's online executions that ensures a strategy exists for the agent to only perform actions that can be extended to a successfully terminating execution of the program, no matter how the environment behaves (Section 7.3).
   - We then show that under some reasonable conditions, if we have sound abstraction and the agent has a strategy for accomplishing a task or achieving a goal at the high level, then we can refine it into a low-level strategy piecewise, and the resulting low-level strategy ensures achieving the refinement of the goal (Section 7.4).

Note that although we approached the work on abstraction of offline and online agent behavior (contributions 2 and 4 respectively) from the point of view of agent supervision, they have many applications beyond this area (e.g., hierarchical classical/contingent planning, agent monitoring, and providing high-level explanations).

1.3 Outline of the Dissertation

The rest of the dissertation is organized as follows: Chapter 2 provides a survey of the related literature. In AI, there does not seem to be many approaches that directly relate to the problems addressed in this dissertation. Hence, we discuss a number of relevant approaches that relate to certain aspects of our work. Chapter 3 lays the theoretical foundations for this dissertation. Chapters 4, 5, 6 and 7 present the main contributions of the dissertation. In Chapter 4, we investigate online supervision of an agent that may acquire new knowledge about her environment during execution and at each time point she must make decisions on what to do next based on what her current knowledge is. In Chapter 5, we formalize a notion of abstracting *offline* agent behavior which is represented by situation calculus action theories, and we discuss the framework's applications in hierarchical planning, providing high-level explanations of low-level agent behavior, and agent monitoring. Chapter 6 builds on the results of Chapter 5 and presents a framework for hierarchical agent supervision in an offline context. Chapter 7 extends the results of Chapter 5 and formalizes a notion of abstraction for *online* agent behavior and its application in hierarchical contingent planning. Finally, in Chapter 8 we conclude with a summary of the contributions of the dissertation and a discussion of future research.

2 Related Literature

In this chapter, we provide a survey of the related literature. In AI, there does not seem to be many approaches that directly relate to our work. Hence, we focus on a number of relevant approaches that relate to certain aspects of our work. Section 2.1 provides a brief overview on reasoning about action and change and two of the main approaches to agent programming in AI. In Section 2.2, we focus on some of the approaches to control and customization of agent behavior. This is followed by the related work on reasoning about knowledge and action in Section 2.3. An overview of some of the semantic and syntactic approaches to abstraction in first order logic is provided in Section 2.4. Finally, in Section 2.5 we review the literature on classical and hierarchical supervisory control of discrete event systems as well as some of the recent approaches in AI that are inspired by this work.

2.1 Reasoning About Actions and Agency

2.1.1 Reasoning About Action and Change

Intelligent agents operating in dynamical domains need to reason about what *actions* to perform to pursue their goals. They also need to keep track of *changes* made to the environment as a result of the actions performed, by themselves or other agents acting in the domain; such changes are governed by specific domain-specific causal laws. Thus, the study of frameworks and formalisms for reasoning about action and change (RAC) is central to the field of knowledge representation and reasoning.

Reasoning about actions and change gives rise to three well-known problems. The first is the *frame problem* [110, 132], which expresses the difficulty to specify and infer all the properties of the world that *do not change* as a result of performing a specific action. For example, paining an object does not cause it to be broken. The second problem is the *ramification problem* [59, 132]. The difficulty is in formalization of all the indirect effects of an action, which can be immense. For instance, painting an object causes all its component parts to be painted as well. Finally, the *qualification problem* [110, 132] expresses the difficulty of effective formalization of all the preconditions of each action, which again can be immense. For example, to paint an object, the object must not be wet, it must be accessible, etc.

A number of logic-based formalisms for RAC exist in the literature, including the situation calculus

[110, 132], the event calculus [90, 149], the fluent calculus [157, 158], temporal action logics (TAL) [52], and action languages (e.g., [67]). The situation calculus is one of the most influential logical formalisms for representing and reasoning about dynamical worlds. In this dissertation, we use Reiter's version of the situation calculus [132], which provides a solution to the frame problem. It is a multi-sorted dialect of first order logic with equality which includes three sorts: *action* for actions, *situation* for situations and a catch-all sort *object* for everything else depending on the domain of application. Actions are assumed to be the cause of all changes in the world, and situations denote a possible world history of sequences of actions performed so far. The changing properties of the world are represented by fluents, which are dynamic predicates or functions that take a situation argument. To represent dynamic domains, Reiter proposed the notion of basic action theories (BATs). These include action precondition axioms which describe when an action is executable (a special predicate $Poss(a, s)$ specifies that action a is executable in situation s); successor state axioms that describe how the value of fluents change; and initial state axioms that describe the initial state (see Section 3.1 for an overview). This framework has been applied in areas such as planning [132], plan recognition [75], customization and control (e.g., [113, 35]), explanation [111], program verification [118], etc.

Reasoning about action and change has not only been central in the field of knowledge representation and reasoning, but it has also played an important role in the more general area of computer science. One may view a computer program as a complex structure whose execution produces a list of actions which the computer performs one after the other, changing the value of variables, and affecting the world (e.g., through an interface). In this way, reasoning about actions provides a means to reason about execution of programs and their effects, and as a result, prove a program's correctness. *Dynamic logic* [81], is a type of modal logic that is intended for verification and reasoning about computer programs that explicitly refers to the actions the computer is executing. Propositional Dynamic Logic (PDL) [81] is one of the best known variants of dynamic logic [164].

A different way of reasoning about programs, typically for non-terminating programs in reactive systems, is expressed in *temporal logic*. This logic is a formalism that augments conventional propositional logic with temporal modalities, making it possible to reason about the ordering of events in time. For example, using temporal modalities *always* and *eventually*, one could assert that "a property p which is not true in the present should *eventually* become true in a future evolution of the system". Or, "a property p should *always* be true in all future evolutions of the system". Such assertions often express *liveness* or *safety* properties of the system. Temporal logic comes in number of flavors such as Linear Temporal Logic (LTL) [128], Computation Tree Logic (CTL) [54], CTL* [55],[3] Alternating-Time Temporal Logic (ATL) [4], μ-calculus [53], etc., and each type has a specific expressive power.

[3]CTL* combines the expressive powers of LTL and CTL.

2.1.2 Agent Programming

Agent programming languages are typically based on theoretical agent frameworks. Such languages include some structure corresponding to an agent, and provide a way to represent attributes of agency such as knowledge, beliefs, goals, commitments or other mentalistic notions [167]. The beginning of the current interest in agent programming languages is a result of Shoham's proposal for *agent-oriented programming* [151].

One of the influential approaches to agent programming is the situation calculus based high-level language of Golog [103] and its successors (e.g., ConGolog [33] and IndiGolog [34]). These languages provide a middle ground between classical planning [120, 121] and programming. The programmer may provide a sketchy non-deterministic program using the domain specific actions and test conditions. Then the interpreter, by using the action theory which includes agent's beliefs about the state of the world, as well as the preconditions and effects of actions, will try to find a provably correct execution of the program. By controlling the level of non-determinism in the program, the high-level program execution task can be made as easy as deterministic program execution or as hard as classical planning. ConGolog [33] extends Golog with concurrency programming constructs and supports exogenous events. IndiGolog generalizes ConGolog with support for sensing and online execution and enables the programmer to control planning/lookahead. We provide an overview on high-level programs based on the situation calculus and online executions in sections 3.2 and 2.3.3 respectively.

Another influential approach to agent programming is based on the BDI (Belief-Desire-Intention) model of agency which is rooted in Bratman's [22] theory of practical reasoning and Denett's [48] theory of intentional systems. Examples of BDI programing languages and platforms include PRS [69], AgentSpeack [18], dMars [51], Jack [163] and 3APL [31].

An important aspect of BDI programming languages and platforms is the interleaved account of sensing, deliberation and execution [68]. Typically, such systems do not perform lookahead or planning in the traditional sense; rather, they rely on programmer defined "plan-libraries" that achieve goals. By executing as they reason, BDI agents aim to make decisions based on updated beliefs and remain responsive to the environment by context-sensitive subgoal selection and expansion. Plan libraries provide computational efficiency, making such systems well-suited for operating in real-time scenarios; however, this approach works well only if good plans can be specified in advance for all objectives that the agent may acquire and all contingencies that may arise.

A recent work by Sardina et al. [143] integrates a BDI agent system with a Hierarchical Task Network (HTN) [120, 121] offline planner as a "lookahead" component in the context of the BDI agent language

CANPLAN. The relation between the BDI based model of agency and situation calculus based programs has also been investigated; for example, Sardina and Lespérance [142] show how to program BDI-style agent systems in IndiGolog.

2.2 Control and Customization of Agent Behavior

2.2.1 Norms

Norms are typically defined as established, expected pattern of behavior of an autonomous agent and have been widely used to regulate and coordinate multi-agent systems. Norms may define what an agent is *obliged* to do or what she is *prohibited* from doing; they may also express temporal constraints such as a *deadline*. Some norms can be regimented, and the agent has no choice but to follow them; in some other cases, it may be possible to violate the norm, and the agent is punished for violating the norm or given a reward for compliance [49].

Alechina et al. [2] present a framework for practical enforcement of regimented norms in multi-agent organizations. A transition system describes the behavior of a (multi-)agent system where each state is labeled with a set of propositions, representing the facts in the world. Norms are specified in LTL, which can be viewed as a set of "good runs" of a transition system which uniquely defines a linear time property. Guards are used to enforce norms; a guard function (characterized by LTL formula with past operators [153]), following a history, can disable specific transitions that (could) violate a norm. Guards are enforced at runtime, thus offering agents more autonomy as disabling of transitions dynamically depends on the history. The authors further define the notion of a *canonical guard*. If a canonical guard can enforce a norm and it is deadlock free, it is considered the optimal guard as it leaves the agents with the greatest degree of autonomy.

It is also shown that even in the presence of unlimited computational power to reason about the future events (i.e., unbounded lookahead) the guard can not enforce certain types of norms (e.g., liveness properties that correspond to obligations without a deadline). The authors further investigate canonical guards for bounded lookahead for liveness and safety properties. An algorithm is presented for computing the minimal window size k such that the canonical guard of a state-based safety norm can enforce the norm. Moreover, a method is provided to compute a lookahead that allows enforcing (although not perfectly or optimally) a liveness norm.

Gabaldon [65] investigates how to express and enforce norms in the Golog programming language. Three types of norms are considered: *ought-to-do norms*, prescribing some actions to be forbidden or obliged; *ought-to-be norms*, prescribing that a state-condition is forbidden; and norms that are a form of deadline.

The norms are represented as formulas, e.g., $\phi(\vec{x}, s)$, with s being the only situation term appearing in the formula. Norms of the same type (e.g., those defining forbidden (resp. obligatory) actions) are re-formatted in a compact normal form. This encoding results in a situation calculus formula that indicates, in any given situation, whether or not an action is forbidden (resp. obligatory). Then, a notion of norm compliant sequence of actions s (i.e., $compliant(s)$) with respect to each type of norm is defined. For example, in case of ought-to-do norms, $compliant(s)$ indicates that a sequence of actions s is compliant with the given norms if and only if s is the empty sequence, S_0, or s is the sequence of actions s' followed by action a, where s' is compliant and satisfies the norms in s'. A Golog program δ is compliant with a set of norms N if all its execution traces comply with the norms (i.e., $\mathcal{D} \models \forall(s', s).Do(\delta, s', s) \supset compliant_N(s)$).

The author then formalizes a method for incorporating a set of norms into the agent's underlying action theory (i.e., domain description) in the form additional preconditions in the action precondition axioms. For example, in case of ought-to-do norms, for an action $A(\vec{x})$, the following conditions are added to the existing precondition axioms: 1) $A(\vec{x})$ is not a forbidden action, and 2) if there is any obligatory action at all, it is $A(x)$. In this way, the agent is guaranteed to behave in a norm compliant manner. Gabaldon also describes notions of equivalence between norm systems with respect to an agent's background theory in the situation calculus, as well as notions of norm system subsumption and consistency.

2.2.2 Control Knowledge

One of the strategies used to decrease the complexity of planning is to employ domain specific constraints often referred to as *control knowledge*. Imposing domain control knowledge on the definition of a valid general plan results in restricting the courses of action (i.e., paths) that achieve the goal. Some influential work in this area include TLPlan [8] and TALPlanner [94] which use control knowledge in the form of declarative constraints expressed in temporal logic formulas.

Gabaldon [64] provides a procedure for compiling search control knowledge into non-Markovian action theories in the situation calculus. In nonMarkovian action theories, successor state axioms and action precondition axioms can refer to situation terms other than the "current" situation s under the restriction that they refer to the past or to an alternative, explicitly bounded future relative to s. Informally, a situation calculus formula is bounded by situation term s if all the situation variables it mentions are restricted, through equality or the $\sqsubset$ (i.e., situation precedence) relation, to range over subsequences of s. A formula that is strictly bounded by s has its situation terms restricted to the past relative to s [62].

In this framework, past temporal logic connectives such as *prev*, *since*, *sometime* and *always* are defined similarly to [62] by using strictly bounded formulas. Since control knowledge is effectively viewed as additional constraints on executability of actions, they are considered to be closely related to the qualification problem. This approach is similar to that of Lin and Reiter's [107] that incorporates state constraints in preconditions of actions. Gabaldon adds control knowledge to the precondition axioms. Given $C(s)$, a formula bounded by s which models some part of the control knowledge, and action $A(\vec{x})$, $C(do(A(\vec{x}), s))$ is added to the preconditions of the action $A(\vec{x})$. By adding the domain closure assumption on actions, a guarantee is provided that a situation is executable only if it satisfies the constraints.

The author also presents a method for transforming a nanMarkovian action theory into Markovian one. The proposed method applies regression steps and introduces additional fluents and their corresponding successor state axioms. This transformation is then used for compiling search control knowledge into normal action preconditions.

2.2.3 Process Customization and Behavior/Service Composition

In *process customization*, a generic process for performing a task or achieving a goal is customized (i.e., refined) to satisfy a *client's specification* (e.g., constraints or preferences) [95]. Process customization includes personalization and configuration and has been applied in numerous areas such as in configuration and specialization of scientific workflows [73], personalization of travel planning [61], customization of the behavioral aspect of software systems [105] and increasing the accuracy of programs generated through using the programming by instruction technique [60].

One may also view *behavior*[4] or *web service*[5] *composition* as a type of process customization, as it involves selecting some available processes and controlling their interaction to produce the desired behavior or service. In AI, automated behavior/service composition has been the focus of many researchers (e.g., [112, 46, 109, 26, 44]) and various techniques such as planning or synthesis have been used to accomplish the task. In this section, we focus on two of the main approaches in the literature.

McIlraith & Son [113] adopt the situation calculus as a theoretical framework for web service composition. Web services are modeled as primitive or complex actions [112] which typically have knowledge and non-knowledge preconditions and effects. In this work, a flexible template for composition in the form of a generic non-deterministic ConGolog procedure is provided. This procedure provides a wide range of alternative ways to perform a task, and during customization, alternatives that violate a given user's constraints are eliminated (and the parameters in the remaining alternatives could be instantiated appropriately). A distinguished fluent ($Desirable(a, s)$) is defined that captures a user's personal constraints with respect to an action a. Then, the computational semantics of the program are updated to ensure the program only makes a transition by executing action a if it satisfies $Desirable(a, s)$ in addition to the domain specific preconditions of a. Moreover, to ensure the agent has sufficient knowledge to execute the program, a notion of *self-sufficient* program is introduced (we discuss this notion in more detail in Section 2.3.4). This work has been extended

[4]A *behavior* stands for the logic of any artifact that can operate in the environment, including web services, agents, software libraries, hardware components, and workflows [44].

[5]Web services are regarded as self-describing, self-contained, modular applications that can be published, located, and invoked across the World Wide Web [125].

[61] to support first-order LTL temporal constraints and preferences. These are compiled into ConGolog and evaluated over a finite horizon. The approach also considers quantitative preferences, and uses DTGolog to maximize the expected utility within the set of most qualitatively preferred plans.

In the "Roman Model" [46, 144, 43, 44] approach to behavior/web service composition, the available behaviors/services (and the shared environment) are abstracted as (non-deterministic) finite state transition systems while the target behavior/service is represented as a deterministic transition system. The composition amounts to synthesizing a controller that orchestrates (i.e., activates, stops or resumes) the concurrent execution of the available modules to "mimic" the desired target behavior/service. The "mimicking" is captured through the formal notion of simulation [82].[6] This work has been extended in a number of directions, such as composition under partial observability [42], and synthesis of a controller generator, i.e., an implicit representation of all controllers, proposed by De Giacomo et al. [44], that returns, at each step, the set of all available services capable of performing the requested action, while ensuring the possibility of delegating to available services all (target-compliant) requests that can be issued in the future.

Yadav et al. [169] extend the "Roman Model" approach by presenting a technique for building the *largest realizable (i.e., supremal) fragment* of a given target specification for behavior composition in the presence of (partially) controllable available behaviors. The behaviors are represented by (non-deterministic) finite transition systems. The notion of *enacted system behavior* is used to refer to the behavior that emerges from the joint execution of available behaviors. The target behavior specification is modeled by another (non-deterministic) finite transition system. A target behavior T_{RTB} is a *realizable target behavior* of a target specification T, if a) T_{RTB} can be simulated by T and b) T_{RTB} can be non-deterministically simulated [144] by the system of behaviors (i.e., there is an exact composition for T_{RTB} on the system).

The technique for obtaining the supremal target behavior relies on two parts: 1) building the synchronous product of the system of behaviors and the target specification; and 2) modifying the obtained structure to enforce conformance on its states which cannot be distinguished by the agent using the target (to request transitions) through introduction of belief level states. This yields the *belief-level full-enacted system.* The authors show that the belief-level full-enacted system represents the unique supremal realizable target behavior.

Then, inspired by supervisory control theory of discrete event systems [166, 24] (see Section 2.5) and reasoning about actions, the composition of behaviors in the presence of uncontrollable exogenous actions is investigated. Such actions are considered to be special events that behaviors may spontaneously generate. The authors propose modifications to the simulation relation and the belief-level full-enacted system that

[6]Intuitively, a transition system $A1$ simulates another transition system $A2$, if $A1$ is able to match any step performed by $A2$ and afterwards, is able to continue to simulate $A2$; see [115] for a formal definition.

enable them to show how the supremal realizable target can again be defined and computed whether the exogenous events are observable or not to the agent using the target.

The "Roman Model" approach has also been investigated in the context of the situation calculus and in the presence of potentially infinite object domains that may go through an infinite number of states as a result of actions [138]. Given a library of available ConGolog programs and a target program not in the library, it is first verified if the target program executions can be realized by composing fragments of the executions of the available programs so as to *mimic* the virtual transitions of the target program at each point in time. The mimicking is realized by using a greatest fixpoint second-order formula based on a suitable notion of simulation [115, 117]. In general, checking such a fixpoint formula over ConGolog programs is undecidable. However, the authors provide a sound procedure that is based on three parts: 1) computation of the simulation through fixpoint approximates, hoping it is possible to compute the fixpoint in a finite number of iterations; 2) use of the characteristic graphs[7] introduced by Clasen & Lakemeyer [29] to finitely cope with the potential infinite branching of ConGolog programs (due to ConGolog's $\pi x.\delta$ construct); and 3) use of regression [132] to compute the preimage of a formula (or weakest precondition). If it is possible to verify the target program executions can be realized by composing fragments of the executions of the available programs, then a delegator is synthesized that does the composition automatically. The authors assume the agent has complete information in the intitial state.

[7]A characteristic graph compactly represents all the possible configurations that a ConGolog program may visit during its execution. Given a ConGolog program, a characteristic graph can be constructed whose nodes represent program configurations (hence a node abstracts a number of program states). Edges in the graph stand for single transitions between program configurations and are labeled with actions and conditions under which these actions can be taken.

2.3 Reasoning About Knowledge and Action

Often, agents do not have *complete* information about their environment, and in many cases, they may be able to acquire new knowledge during execution. In this section, we first discuss some approaches to *sensing* in the situation calculus, which allows the agent to acquire new knowledge from the environment; we then discuss how *knowledge* may be represented and affected (by sensing actions) in the situation calculus. In Section 2.3.3 we look at some approaches to *online execution* of an agent in the situation calculus, where executions are considered to allow for interleaving sensing and deciding what action to execute next. We then look at some approaches in the situation calculus that model *ability*, which is typically used to refer to the agent having the necessary knowledge to achieve a goal, to execute a plan, or to achieve a goal by executing a given plan. Finally, we discuss some approaches to conditional planning in the situation calculus, which provides a way to plan in presence of uncertainty.

2.3.1 Sensing

In the situation calculus, *sensing actions* provide an agent with new knowledge about her environment and as such, affect her *mental* state, leaving the world unchanged otherwise. For instance, by performing the action of sensing if a certain road is closed, the agent comes to *know* whether the road is closed. Thus, such actions are often called *knowledge-producing actions* [132]. Various ways of modeling sensing in the situation calculus have been proposed; in the following we look at the main approaches.

Levesque [101] introduces a special fluent $SF(a, s)$ (for *sensed fluent value*). He further formalizes axioms that describe how the truth value of the fluent SF becomes correlated with those aspects of a situation which are being sensed by action a. For instance, the axiom $SF(senseRoadClosure(r), s) \equiv Closed(r, s)$ states that when the action $senseRoadClosure(r)$ is executed, some sensor returns a value, which then informs the agent whether road r is closed in situation s. Scherl and Levesque [145] show how non-binary sensing results, such as reading a temperature which returns a numerical value, can also be represented within the framework.

DeGiacomo and Levesque [40] generalize Levesque's approach [101] and allow conditional sensing axioms. It is assumed that the agent has a number of sensors that provide sensing readings at any time. The fluent SF is replaced by a finite number of *sensing functions.* These are unary functions whose only argument is a situation (e.g., $roadClosure(s)$). A *sensor-fluent formula* is defined to be a formula of the language, that does not include $Poss$, and uses at most a single situation term, which is a variable; this term only appears as the final argument of a sensor function or fluent. A *sensor formula* is a sensor fluent formula that does not include any fluents. A *guarded sensed fluent axiom* (GSFA) is defined as $\alpha(\vec{x}, s) \supset [F(\vec{x}, s) \equiv \rho(\vec{x}, s)]$, where α is a sensor-fluent formula referred to as the *guard* of the axiom, F is a relational fluent, and ρ is a sensor formula. An action theory may contain any number of GSFAs for each fluent. For example, $nearRoad(r,s) \supset Closed(r,s) \equiv roadClosure(s) = 1$ indicates that when the agent is near road r, and the result of sensing function $roadClosure(s)$ is 1, then the road r is closed.

Lespérance et al. [99] on the other hand, represent sensing as an ordinary action which queries a sensor, followed by the reporting of a sensor result, in the form of an exogenous action. For example, to sense whether fluent $Closed(r,s)$ holds within a ConGolog program, the macro $senseRoadClosure(r) \doteq qryRoadClosure(r); (report(r,1) \mid report(r,0))$ is defined, where $qryRoadClosure(r)$ is an ordinary action that is always executable and is used to query (i.e., sense) if the road r is closed and $report(r,x)$ is an exogenous action with no effect on the state/fluents that informs the agent if $Closed(r,s)$ holds through its precondition axiom, which is of the form $Poss(report(r,x),s) \equiv Closed(r,s) \wedge x = 1 \vee \neg Closed(r,s) \wedge x = 0$. In this way, the agent learns that action's preconditions must have held and that the road must have been closed if and only if $x = 1$ (if she did not know this information already).

2.3.2 Knowledge

In many dynamical systems where an agent has incomplete information about her environment, it is necessary to formalize an explicit notion of the agent's knowledge as well as the effects of knowledge-producing actions on her mental state. Moore [119] was the first to provide a possible-world semantics[8] for a logic of knowledge in the situation calculus by considering situations as possible worlds.

Based on Moore's work, Scherl and Levesque [145] introduce a new binary predicate $K(s',s)$ which is an accessibility relation over situations; $K(s',s)$ indicates that situation s' is accessible from situation s, meaning that as far as the agent knows in situation s, she might be in situation s'. An agent *knows* a proposition ϕ in the situation s, i.e., $Knows(\phi,s)$, if ϕ is true in all K-accessible situations: $Knows(\phi,s) \overset{\text{def}}{=} \forall s'.K(s',s) \supset \phi[s']$. Novel to Scherl and Levesque's approach, is the successor state axiom for fluent K that captures the effect of actions on knowledge, while providing a solution to the frame problem:

$$K(s', do(a,s)) \equiv \exists s''.s' = do(a,s'') \wedge K(s'',s) \wedge Poss(a,s'') \wedge [SF(a,s'') \equiv SF(a,s))]$$

Intuitively, the axiom states that after performing an action a in situation s, the agent thinks she may be in situation s' if and only if, s' is the situation resulting from executing action a in some situation s'' which is K-accessible in s, assuming that action a is executable in s'' and s' and s agree on the value of SF (i.e., the

[8]The possible worlds semantics for modal logics had earlier been proposed by different researchers including Kripke [91] and Hintikka [83]; the latter dealt with epistemic logic.

property being sensed). The effect of this axiom is to remove from consideration those accessible situations where the sensed property has a different value. To accommodate the notion of epistemic alternatives, the foundational axioms of basic action theories are adapted to allow for multiple initial situations. It is further shown that if the K relation on the set of initial situations is restricted to conform to some important general properties of accessibility relations like reflexivity along with some subset of the symmetric, transitive and Euclidean properties, then the K relation at every non-initial situation, will satisfy the same set of properties.

In the approaches by Lespérance et al. [99] and DeGiacomo and Levesque [40] an explicit notion of knowledge is not represented in the language; instead, the logical consequences (entailments) of the basic action theory are used indicate what the agent knows about its environment. Such approaches avoid the complexity of modeling knowledge explicitly, but at the same time, are not able to represent introspection.

2.3.3 Online Executions

High-level programs defined over the situation calculus, such as Golog and its successors [103, 33, 140] enable a user to specify an agent's behavior by providing a sketchy non-deterministic program. In *offline executions* [39] the interpreter must find a sequence of actions constituting an entire legal execution of a program *before* actually executing any of actions in the world. In *online executions* [39, 140, 34] instead, incremental executions of the program are considered that allow for interleaving sensing and deciding what action to execute next.

De Giacomo et al. [39, 34] and Sardina et al. [140] present a framework for online executions that uses the SF fluent to specify sensing. To describe what the agent has learned so far, the notion of a *history* σ and the formula $Sensed[\sigma]$ are used. A history $\sigma = (a_1, \mu_1) \dots (a_n, \mu_n)$ is a sequence of actions performed (a_i) together with their sensing results (μ_i which can be 0 or 1). SF is specified by axioms as mentioned earlier. $Sensed[\sigma]$ is the conjunction of sensed information obtained in history σ. For example, $Sensed[(a, 1), (b, 0), (c, 1)] = SF(a, S_0) \wedge \neg SF(b, do(a, S_0)) \wedge SF(c, do(b, do(a, S_0)))$.

The program execution uses the theory together with results of sensing accumulated so far to decide if it has already reached its goal and can terminate, take a step of the program by executing a single action, or take a step of the program with no action needed. After the agent performs an action, any new sensing results provided by this action are gathered, and the execution process iterates. Unlike offline executions where the reasoner may need to determine lengthy course of action before executing the program, online executions utilize the sensing information provided by the first n actions performed in deciding the $(n+1)$th action. However, once the agent performs an action, and later in the execution it becomes clear that a non-deterministic choice was resolved incorrectly, there may be no way to backtrack. As a solution, a new programing construct, namely the search operator Σ_s is introduced in [39, 34]. Given a program δ, the program $\Sigma_s(\delta)$ executes online in a similar way that δ does offline. Hence, before committing to any action, it first ensures by offline reasoning that this action can be extended to a complete execution of δ. Being able to search in an offline manner in an online system presents a challenge of finding the right balance between offline reasoning which provides a safer execution but more downtime, and online responsiveness.

In the approach by Lespérance et al. [99] mentioned earlier however, the sensing results are stored in the situation (which includes all actions executed so far including the exogenous actions used for sensing. $Executable(do(\vec{a}, S_0))$, which means that the preconditions of every action performed in sequence of actions $\vec{a}$ were satisfied in the situation in which it occurred, captures what the agent has learned so far during execution (instead of $Sensed[\sigma]$).

A challenge in online executions is that an agent's predicted mental world (as determined by her theory) may not always conform to the actual state of the world (as indicated by sensing results). Execution monitoring [45] and process adaptation [47] are two approaches that investigate how to resolve such discrepancies.

A middle ground between offline and online execution has also been investigated by McIlraith and Son [113]. In this approach online sensing collects the relevant information while the effects of world-altering actions are simulated offline, and if successful, carried out. This approach operates under the assumption of reasonable persistence of sensed information, which in some scenarios, may not hold.

2.3.4 Ability

In the literature on agent theories, *ability* has been used to refer to the agent having the necessary knowledge to *achieve a goal*, to *execute a plan*, or *achieve a goal by executing a given plan* [70]. The notion of ability has been studied under different names such as "epistemic feasibility", "self sufficient programs", "knowledge prerequisites of actions", "knowing how to execute a program", "ability to achieve a goal", etc. Moore [119] was one of the first to consider the concept of ability. He formalized a theory that integrates knowledge and action into a single framework using Hintikka's modal logic of knowledge [83] and a modal adaptation of McCarthy's situation calculus [110]. Based on Moore's approach to formalizing ability and using McCarthy's situation calculus, Davis [32] provided a formal account of ability. These approaches do not address the frame problem. Moreover, Davis does not consider the ability to achieve a goal. Van der Hoek, van Linder, and Meyer [160] on the other hand, proposed a propositional modal logic of ability. We would also like to mention the approach by Singh [152], which formalized a framework for a logic of situated know-how based on a propositional branching time temporal logic. In the following we look at several influential approaches to defining ability that are based on Reiter's situation calculus.

Lespérance et al. [100] formalize an account of knowing how to execute a plan and ability to achieve a goal for an agent that can acquire information about her environment at runtime. This framework is based on the Scherl and Levesque's model of knowledge [145].To enable an agent to know which action to execute in each situation to achieve a goal, a notion of *strategy* is introduced. A strategy is defined as an action selection function, i.e., a mapping from situations to primitive actions. The predicate $CanGet(\phi, \sigma, s)$ denotes that the agent, starting in situation s, knows at every step the action prescribed by strategy σ so that she "can get" to a situation where she knows the goal ϕ holds. The agent "can" achieve a goal ϕ in situation s, i.e., $Can(\phi, s)$, if and only if there exists a strategy σ such that she knows in s that she "can get" to a situation where the goal holds by following σ. This framework is shown to handle both ability and inability of agents in cases involving unbounded iteration.

The authors further formalize the notion of knowing how to execute a plan represented by a Golog program. A path selection function is defined that specifies a deterministic execution strategy over a non-deterministic Golog program. The agent "can execute" a program δ following a given strategy σ in s, denoted by $CanExec(\delta, \sigma, s)$ if and only if the program terminates when executed according to σ and at every point during the execution, the agent knows whether the program has terminated and if not, she knows which action to execute next. Based on this definition, the notions of *dumb know-how* and *smart know-how* are introduced. In dumb know-how, the agent may arbitrarily pick any action during execution of a non-deterministic program, as she is able to execute the program according to *all* strategies to ensure successful execution. This notion is particularly useful in cases where an agent wishes to delegate a task to another agent; in this case the other agent only has to execute the program. On the other hand, in smart know-how, the agent must look ahead before committing to any execution strategy; thus it is only required that *there exists* a strategy that the agent knows she can follow to execute the program.

The relation between the notions of ability to achieve a goal and knowing how to execute a plan is also investigated; an agent is able to achieve a goal ϕ if and only if she smartly knows how to execute the program $while\ \neg Know(\phi)\ do\ \pi a\ a;$, which intuitively means while she does not know that ϕ holds, she can non-deterministically choose any action and execute it. By considering ahead of time whether there are alternatives at every choice point whose choice guarantees that she will be able to complete the execution of the program, she can ensure a successful execution.

Sardina et al. [139] develop a non-epistemic formalization of deliberation[9]/planning under the assumption of incomplete information and sensing. This framework is based on the situation calculus and ConGolog. Two approaches for defining when an agent knows how/is able to execute a deterministic program δ in a history σ are considered: an Entailment-Consistency (EC)-based knowing how and an Entailment-Truth (ET)-based

[9]Deliberation refers to reasoning about possible execution strategies.

knowing how.

In the EC-based approach, $KHow_{EC}(\delta, \sigma)$ is defined as the smallest relation that contains all the program configurations that are either terminating, or from which an action transition exists such that for every consistent sensing outcome, the resulting configuration is in $KHow_{EC}(\delta, \sigma)$. The relation $KHow_{EC}(\delta, \sigma)$ uses entailment to ensure that the available information is sufficient to determine which transition should be performed next; consistency is used to determine which sensing results may occur, for which the agent should have a sub-plan that can lead to a final configuration. It is then shown that the EC based account of knowing how fails on some programs involving unbounded iteration. The problem results from the local consistency checks that determine which sensing results are possible; the models that satisfy the overall domain specification, and those that do not, can not be distinguished. On the other hand, whenever $KHow_{EC}(\delta, \sigma)$ holds, a finite/bounded conditional plan (i.e., a tree program) exists that can be followed to execute δ in σ. Note that Moore's account of ability is closely related to $KHow_{EC}$ and thus inadequate for dealing with unbounded iteration.

In the ET-based account of knowing how, models of the environment are fixed (although it is not known exactly which model represents the actual environment). $KHow_{ET}(\delta, \sigma, M)$ is defined similarly to $KHow_{EC}(\delta, \sigma)$, with the difference that the sensing results come from the fixed model M. Given this definition, an agent knows how to execute a program δ in σ, i.e., $KHow_{ET}(\delta, \sigma)$, if and only if for every model M that satisfies the theory and the sensing results accumulated so far, $KHow_{ET}(\delta, \sigma, M)$ holds. Thus, the ET based account is able to handle cases of programs with unbounded iteration.

In related work, Sardina et al. [141] identify a fundamental limitation with planning over belief states under incomplete information. In such settings, the actions of a planning problem are typically modeled as non-deterministic transitions over the belief states of the planner. A plan is considered *adequate* if it works from the initial belief state, and, where a plan is considered to work from some belief state if and only if either (1) it indicates no action should be taken and the goal is believed to hold, or (2) it indicates some action should be taken, and for every possible belief state that can be reached by doing the action, the remaining plan works in the resulting state. It is then shown that such view of plan adequacy requires an upper bound of the number of actions performed, thus, making it impossible to work for iterative planning.

The authors then suggest viewing actions as deterministic transitions over world states while applying plans to belief states. The belief state is considered to be a non-empty set of the world states. A procedure for inducing a belief-based planning problem from a world-based planning problem is provided which includes a definition of $k(w, a, b)$. This definition is similar to the successor state axiom of [145], where w represents the real world, b the set of worlds accessible from w and $k(w, a, b)$ models the new set of accessible worlds after doing action a in world w and belief state b.

The world-based version of planning is used to show why unbounded plans are considered as not adequate. Based on definition of the adequate plan, at any point during execution, the plan must work under the assumption that sensing results may come from any member w of the current belief state b. Thus, until it becomes known that the goal has been reached, there will always be elements of b where more actions need to be executed to reach the goal. This in turn results in a situation where no progress seems to be happening in the real world.

To resolve this issue, each world state in the initial belief state needs to be considered separately, to see if the plan will work in each case. The definition for plan adequacy is then adapted to consider sensing with respect to a world state w (that changes systematically as actions are performed) instead of arbitrary elements of the current belief state b.

McIlraith and Son [113] formalize a *self sufficient* (ssf) property of Golog programs, which indicates that at each step of program execution, the agent has all the required knowledge to execute that step. A Golog program δ is self sufficient in situation s, denoted by $ssf(\delta, s)$ if and only if the agent has the necessary knowledge to execute δ in s. ssf is defined inductively over the structure of δ. For example, $ssf(\varphi?, s) \equiv KWhether(\varphi, s)$, indicates the test action φ is considered self sufficient in situation s if and only if $KWhether(\varphi, s)$ holds, i.e., the truth value of φ is known in situation s [145]. The authors also identify syntactic accounts of self-sufficiency; for instance, programs in which each conditional or loop construct that conditions on φ is preceded by a sensing action that ensures the truth value of φ is known prior to such constructs and persists until it is queried. This approach is incomplete for programs that involve indefinite iteration.

2.3.5 Conditional Planning

Conditional planning [120, 121] provides a way to plan in presence of uncertainty, making it possible to deal with any contingencies that may arise. The sources of uncertainty include partial observability, incomplete information in initial state and non-deterministic effects of actions.

Several approaches have studied planning under incomplete information and sensing. These include PKS [126], a knowledge-based planner based on generalization of STRIPS; planning based on model checking [26]; and a dynamic programming method for computing belief-based policies and a heuristic search method for computing history-based policies proposed by Geffner and Bonet [66]. Reasoning about strategies in presence of incomplete information in the situation calculus has also been investigated [168].

Lespérance et al. [99] develop a formal model of contingent planning for an agent operating in a dynamic and incompletely known environment. The agent's task and the behavior of other agents in the environment are represented as concurrent high-level non-deterministic programs δ and ρ respectively. Programs can belong to any agent programming language with a transition semantics. It is assumed that ρ runs at a higher priority than δ, and it is further required that in each state, either the agent or the environment can execute an action. Moreover, the agent program can perform an action when it *knows* (i.e., it is entailed by its belief base) that there is legal transition involving that action to the next program configuration; the environment actions may occur if they are consistent with agent's belief base. The environment (exogenous) actions are considered to be fully visible. Upon observing any environment action, the planning agent can learn that the action must have been executable; in other words, its preconditions held. In this way, any environment action can be considered as "knowledge-producing". In such a context, the planner is required to provide the agent with a deterministic conditional plan that can be successfully executed against every possible execution of the environment. A number of restrictions are imposed on the environment program, for example, in any configuration, it should be *known* what the environment may do next.

In this framework, $Able(\delta, \rho, s)^{+}$ denotes that the agent is able/knows how to execute δ in an environment with behavior specified by ρ in state s, and is defined as the smallest relation $R(\delta, \rho, s)$, which holds for all triples (δ, ρ, s) when 1) the environment can not execute any actions and the agent program is final, 2) the agent can execute *some* action that leads it to a configuration (δ', s') where $R(\delta', \rho, s')$ holds, and 3) for *all* the finite sequences of transitions the environment may perform leading to configuration (ρ', s'), $R(\delta, \rho', s')$ holds.

$Able(\delta, \rho, s)^{+}$ can be used to generate a conditional plan σ for the agent to follow to successfully execute its program no matter how the environment behaves. $AbleBy(\sigma, \delta, \rho, s)$ is defined as the smallest relation $R(\sigma, \delta, \rho, s)$ such that 1) for all triples (δ, ρ, s), if the environment is blocked and the agent program is final, then $R(nil, \delta, \rho, s)$ holds; 2) for all quadruples $(\sigma, \delta, \rho, s)$, if the agent program can make a transition to configuration (δ', s'), and $R(\sigma, \delta', \rho, s')$ where $s' = s$ (i.e., the agent program performed a test action), then $R(\sigma, \delta, \rho, s)$; 3) for all quadruples $(\sigma, \delta, \rho, s)$, if the agent program can execute *some* action a that leads it to a configuration (δ', s') then a is prefixed to the existing strategy σ, and $R((a; \sigma), \delta, \rho, s)$ holds; and, 4) for all triples (δ, ρ, s), if for *all* the finite sequences of transitions the environment may perform leading to configuration (ρ', s') where the environment becomes blocked, implies for some σ', $R(\sigma', \delta, \rho', s')$ holds, then $R(\sigma, \delta, \rho, s)$ holds where σ is adapted to include a sub-strategy for each possible action by the environment. This framework does not support goals which require plans with unbounded iteration. A concrete formalization of this framework for the ConGolog agent programming language is also defined.

2.4 Abstraction

2.4.1 Theoretical approaches to abstraction in AI

Logical theories of abstraction are oriented towards theorem proving and have typically involved either a (mainly) *syntactic* or *semantic* approach to abstraction. While syntactic theories consider abstractions defined as mappings between logical languages that do not deal with models of theories, semantic abstractions are primarily concerned with the correspondence between models. In this section we provide an overview of some of the influential approaches that focus on first-order formal systems.

2.4.1.1 Syntactic Theories of Abstraction

Plastid [127] was the first to propose a general theory of abstraction focused on theorem proving and in particular on resolution. He viewed abstraction as mapping from a set of clauses B to a simpler (i.e., more abstract) sets of clauses A. Given a problem in B, the solution corresponds to a solution in A, but A may have additional solutions. Thus solutions in A are used as a guide to search for solutions in B.

An abstraction is defined as an association of set $f(C)$ of clauses with each clause C such that: 1) if clause C_3 is a resolvent of clauses C_1 and C_2, and moreover, $D_3 \in f(C_3)$, then there exists $D_1 \in f(C_1)$ and $D_2 \in f(C_2)$ such that some resolvent of D_1 and D_2 subsumes D_3; 2) $f(Nil) = \{Nil\}$; and, 3) if C_1 subsumes C_2, then for each abstraction D_2 of C_2 there exists an abstraction D_1 of C_1 such that D_1 subsumes D_2. If f is a mapping that satisfies these properties, then it is called an abstraction mapping of clauses. Moreover, if $D \in f(C)$ then D is called an abstraction of C. Plastid then shows how an abstraction mapping between clauses and sets of clauses can be obtained from a mapping between the literals.

Plastid provided several examples of abstraction mappings, such as *propositional abstraction*, *deleting arguments* and *renaming predicate and function symbols*. In the latter, several predicates (resp. functions) could be renamed to the same predicate (resp. function) in the abstract clause. Plastid was aware that this approach could create an inconsistent abstract space from consistent ground space (e.g., renaming both predicates p_1 and p_2 to the same predicate p, and assuming that the ground space satisfies p_1 and $\neg p_2$) and he called this issue the "false proof" problem.

Tenenberg [156] focused on *predicate abstractions* which can be considered as a special case of *renaming predicate and function symbols*. Predicate abstractions view distinctions among a set of predicates $P_1, \ldots, P_n$ as often irrelevant with respect to a certain reasoning task. Tenenberg introduced *Restricted Predicate Mapping*, and defined abstraction at a syntactic level as a mapping between predicates, where among all correspondences between predicates in the ground and abstract spaces, only those that preserve consistency are kept. For example, suppose the concrete level includes the predicates p_1 and p_2, and that we wish to map them both to predicate p in the abstract level. Suppose further that the low-level theory includes the following: $p_1(x) \supset a(x)$, $p_2(x) \supset a(x)$, and $p_2(x) \supset b(x)$. One solution is to remove from mapping all those predicates in the original theory that distinguish among p_1 and p_2, in this case, b. So the abstract theory could only contain $p(x) \supset a(x)$. This solution may be stronger than required, so a weakening of the requirement is to allow abstractions such as $p_1(x) \vee p_2(x) \supset p(x)$.

A highly influential approach by Giunchiglia and Walsh [74] considers abstraction as a mapping between two representations of a problem modeled as formal systems. A formal system (i.e., a formal description of the theory) is defined as a pair $\Sigma = (\Theta, L)$, where Θ is a set of well-formed formulas (wffs) in the language L. Typically the language L is defined by the alphabet, the set of well-formed terms, and wffs; however, for simplicity, it is assumed the language is the set of wffs; the alphabet and well-formed terms are given implicitly by providing the set of wffs.

An abstraction $f : \Sigma_{base} \Rightarrow \Sigma_{abs}$ is defined as a mapping between formal systems Σ_{base} (i.e., the *ground* space) and Σ_{abs} (i.e., the *abstract* space), with languages L_{base} and L_{abs}, respectively, and an effective, total function $f_L : L_{base} \rightarrow L_{abs}$ which is referred to as the *mapping function.*

In a formal system, $TH(\Sigma)$, the set of *theorems* of Σ represents the minimal set of wffs, including the axioms, that is closed under the inference rules. As the authors are focused on theorem proving, they investigate how an abstraction affects provability; that is, when Θ is $TH(\Sigma)$. The abstraction mappings are classified with respect to provability preservation: the set of theorems $TH(\Sigma_{abs})$ can be equal, a subset or a superset of the abstractions of the theorem set $TH(\Sigma_{base})$. These are referred to as Theorem-Constant (TC) abstraction, Theorem-Decreasing (TD) abstraction and Theorem-Increasing (TI) abstraction respectively. TI abstractions can generate false proofs.

Giunchiglia and Walsh classify different uses of abstraction in theorem proving, along two main dimensions of *deductive* versus *abductive*, and *positive* versus *negative.* For TD abstractions, with the combination of deductive/positive, if it can be proved that an abstract wff is a theorem ($f_L(\alpha) \in TH(\Sigma_{abs})$), then it is *guaranteed* that the ground wff is a theorem ($\alpha \in TH(\Sigma_{base})$). However, with the combination of abductive/negative, if it can not be proved that an abstract wff is a theorem, then it may be the case that the ground wff may not be one as well. For TI abstractions, with the combination deductive/negative, if it cannot be proved that an abstract wff is a theorem, then the ground wff is not a theorem. However, with the combination of abductive/positive, if it can be proved that an abstract wff is a theorem, it may be the case that the ground wff may be a theorem as well.

The authors also investigate when two abstractions are considered equal and when one abstraction is stronger than the other. A notion of composition of abstractions is also studied. Giunchiglia and Walsh reconstruct a number of influential approaches to abstraction, such as planning with ABSTRIPS and predicate abstraction of Plastid and Tenenberg in their framework and analyze their properties. For example, Plastid's approach is considered a TI abstraction while Tenenberg's restricted predicate mapping is viewed as TD abstraction.

The syntactic theory of abstraction captures important aspects of different types of abstractions. Many abstractions are a result of manipulating formulas syntactically. Also, as theorem provers reason by applying inference rules to formulas, understanding the properties of abstractions as mappings among formulas is essential [122]. On the other hand, a major shortcoming of the syntactic theory of abstraction is that while it captures the final result of an abstraction, the theory does not explicitly capture the underlying justifications or assumptions that lead to the abstraction, nor the mechanism that generates the abstraction itself [122, 137]. We look at some semantic approaches to abstraction that address this shortcoming in the following section.

2.4.1.2 Semantic Theories of Abstraction

Ghidini and Giunchiglia [72] propose a semantic formalization of the notion of abstraction that is based on the Local Models Semantics [71]. The semantics captures the two main intuitions underlying contextual reasoning: (1) reasoning is mainly local and uses only part of what is potentially available (e.g., partial view of the system), this part is called *context* (of reasoning); at the same time, (2) there is compatibility among the reasoning performed in different contexts.

Abstraction is defined as a mapping function $f_{abs} : L_{base} \to L_{abs}$ between a ground language L_{base} and an abstract language L_{abs}. This function is total, effective and surjective; moreover, it only maps atomic formulas in the languages. The abstract (resp. ground) language has a set of models M_{abs} (resp M_{base}) which consist of *local* models. Then, a *domain relation* is defined that represents the relation between domains of the ground and abstract models. All domain relations are assumed to be total and surjective functions. It is further assumed that all local models in each set agree on the interpretation of terms, but may differ in interpretation of predicates. A *compatibility pair* represents either a pair of local models in M_{base} and M_{abs} or the empty set $\emptyset$. The abstraction mapping is represented by a *compatibility relation*, which is a set of compatibility pairs; the compatibility relations link what is true in the two sets of models. In this way, a model of an abstraction function is a set of pairs of models, which are models of the ground and abstract languages.

Nayak and Levy [122] propose a semantic theory of abstraction as a *model level* mapping. They view abstraction as a two-step process: in the first step, the intended domain model is abstracted; and in the second step, a set of abstract formulas is constructed that capture the abstracted domain model.

Given a base language L_{base} and an abstract language L_{abs}, an *abstraction mapping* π is defined as a mapping between interpretations of the two languages: $\pi : Interpretations(L_{base}) \rightarrow Interpretations(L_{abs})$. The authors further define the notion of *Model Increasing (MI)* abstractions. Given T_{abs} and T_{base} are sets of sentences in L_{abs} and L_{base} respectively, and π is an abstraction mapping, T_{abs} is an MI abstraction of T_{base}, relative to π, if for every model M_b of T_{base}, $\pi(M_b)$ is a model of T_{abs}. MI abstractions are a subset of TD abstractions and thus do not generate false proofs.

The mapping π is characterized in terms of *interpretation mappings* [56]. To specify the abstraction mapping π, one must find the appropriate formulas in the base language L_{base} that define how the abstract universe and denotations for abstract object, function and relations are constructed from a base model. More formally, an interpretation mapping π that maps a model M_b of T_{base} to an interpretation of $\pi(M_{base})$ of L_{abs} consists of:

(1) a wff $\pi_\forall$ with a single free variable v_1. Given any model M_{base} of T_{base}, $\pi_\forall$ formalizes the universe of $\pi(M_{base})$ as the set defined by $\pi_\forall$ in M_{base}

(2) for every n-ary relation R_{abs} in L_{abs}, a wff $\pi_{R_{abs}}$ with n free variables $v_1, \ldots, v_n$ that defines R_{abs}. More specifically, given any model M_{base} of T_{base}, $\pi_{R_{abs}}$ formalizes an n-ary relation in M_{base}. Moreover, the denotation of R_{abs} in $\pi(M_{base})$ is this relation restricted to the universe of $\pi(M_{base})$

(3) similar wffs are used to specify denotations of abstract functions and objects [56]

MI abstractions have a number of properties. For instance, given that T_{abs} is an MI abstraction of T_{base}, if T_{abs} is inconsistent, then T_{base} is also inconsistent. As another example, MI abstractions support compositionality: given that T_{abs} is an MI abstraction of T_{base} with respect to to π and S_{abs} is an MI abstraction of S_{base} with respect to π, then $T_{abs} \cup T_{base}$ is an MI abstraction of $T_{base} \cup S_{base}$ with respect to to π. The authors further present the notion of *strongest* MI abstraction of a base theory, which is the result of an abstract theory *precisely* implementing the intended model level abstraction. A procedure based on resolution is provided that constructs the strongest MI abstraction for a given model level mapping. In this procedure, the authors focus on the case where L_{abs} results from adding a set of new predicates and removing some old predicates from L_{base}; the object and function constants are unchanged and the abstract language is assumed to not include equality.

In Model theory [84], an interpretation of a structure A_1 in another structure A_2 (whose signature may be unrelated to A_1) is a notion that approximates the idea of representing A_1 inside A_2. Interpreting structure A_1 in a structure A_2 results in the ability to translate every first-order statement about A_1 into a first order statement about A_2; this implies the complete theory of A_1 can be read from that of A_2. If it is possible to generalize this notion to interpreting a family of models of a theory T_1, always using the same defining formulas, then the resulting structures will all be models of a theory T_2 that can be read from T_1 and the defining formulas. This can indicate when the theory T_1 is interpreted in the theory T_2. This notion can be used to indicate if one theory is reducible to another.

2.4.2 Hierarchical Planning

Planning [120, 121] is one of the fields in which abstraction has played an important role since the beginning. In this context abstraction typically involves transformation of a problem representation to one which allows the problem to be solved with reduced computational effort. Thus, often a hierarchical representation is generated that corresponds to different levels of abstraction and cost reduction [137].

In the planning literature, different notions of abstraction have been investigated. These include *precondition elimination abstraction*, first introduced in the context of ABSTRIPS [135]; *task networks* in *Hierarchical Task Networks* (HTNs) (e.g., [170, 57]), which abstract over a set of (non-primitive) tasks; and *macro operators* (e.g., [89]), which represent meta-actions built from a sequence of action steps. Another recent approach is *generalized planning* [86], where a general solution for a problem class can be used to solve any particular instance of the class. Aguas et al. [1] propose hierarchical finite state controllers for *generalized planning* that can solve a range of similar planning problems.

McIlraith and Fadel [112] investigate planning with *complex actions* (a form of macro actions) specified as Golog [103] programs. In this work, the authors focus on terminating deterministic complex actions. To enable a planner to compose both primitive and complex actions to achieve a goal, preconditions and successor state axioms for complex actions are defined. Abbreviation $do_{ca}(\delta, s)$ denotes a situation resulting from performing complex action δ in s. For each complex action δ, $Poss_{ca}(\delta, s)$ denotes its preconditions which are intuitively defined in terms of the preconditions of all the actions that make up the execution of δ, (i.e., $Poss_{ca}(\delta, s) \equiv \exists s.Do(\delta, s, s'))$. Moreover, the effects of a complex action δ are assumed to be effects of each action in the execution of δ modulo the effects of subsequent actions. Technically, since complex actions involve multiple intermediate situations, successor state axioms for complex actions can not be defined, and instead, pseudo-successor state axioms are defined. The truth of fluent F after performing δ is defined as $Poss_{ca}(\delta, s) \rightarrow [F(\vec{x}, do_{ca}(\delta, s)) \equiv \exists s.Do(\delta, s, s') \wedge F(\vec{x}, s') \wedge s' = do_{ca}(\delta, s)]$. To enable planning with complex actions as operators, the preconditions need to be characterized strictly in terms of the situation in which the complex action execution is initiated, and effects, in terms of initiating and terminating situation of the complex action. To achieve this, the authors define a new version of the regression operator [132], that regresses over the successor state axioms for the primitive actions in the theory. The defined regression operator is used to regress over the right hand side of the action precondition axioms for each complex action, and the right hand side of the pseudo-successor state axioms for each fluent-complex action pair.

Given an action theory T_A and a set of complex actions Δ_A, first Δ_A is compiled into the theory resulting in a new theory T'_A where each complex action is represented as a new primitive action. After producing a plan in T'_A, the theory is re-written and expanded to replace complex actions with a sequence of primitive actions. The authors further provide some experimental results which shows that in some domains, planning with complex actions can improve the efficiency of planning.

Gabaldon [63] provides an encoding of totally ordered HTNs in Golog. The HTN planning problem is defined as $P = (d, S_0, D)$, where d is a task network, S_0 is the initial situation, and D is a planning domain consisting of set of methods in addition to a situation calculus basic action theory.

Primitive tasks are represented as primitive actions. For each compound task, with a set of methods, a Golog procedure is defined in terms of non-deterministic choice of all the methods. The procedure non-deterministically chooses one of the methods to execute, and then the sequence of precondition tests and tasks relevant to that method are executed. Δ_P is used to denote the resulting set of procedure declarations. Furthermore, a Golog program δ_d is obtained from the task network d which has the same form as the procedures defined for each method. Given the above encoding, a logical specification of the planning problem in terms of Golog is $D \models (\exists s) Do(\Delta_P; \delta_d, S_0, s)$, which means that the theory entails there is an execution of Δ_P followed by δ_d which terminates in situation s. Gabaldon further shows that the operational semantics of an HTN-planning problem, i.e., the set of solutions, corresponds to the set of execution traces derived from the situation calculus formalization i.e., $\sigma \in solutions(d, S_0, D)$ if and only if $D_P \models Do(\Delta_P; \delta_d, S_0, do(\sigma, S_0))$. A similar approach is taken for defining *partially ordered* HTN in ConGolog which supports concurrent execution of procedures. The author also shows how advanced features such as sensing and exogenous actions can be utilized in this encoding.

2.4.3 Conditional Planning with Abstraction

Different approaches to hierarchical planning in presence of incomplete information have been suggested. These include conditional HTN planners [93, 3], and an approach by Srivastava [155] based on model checking techniques [136] for abstracting collections of states with different objects quantities and properties.

Baier and McIlraith [10], building on the results of [112], investigate planning with sensing where both primitive and complex actions are used as building blocks of the plan. This framework is based on the situation calculus and the Golog programming language and uses the knowledge model of Scherl and Levesque [145]. The authors focus on deterministic tree programs. To ensure that at each step of program execution, the agent has the necessary knowledge to execute that step, the programs are required to satisfy the self-sufficiency property [113] (see Section 2.3.4). The authors propose an offline execution semantics for Golog programs with sensing that incorporates the knowledge of the agent. In this approach, the right hand side of the transition semantics ($Trans$) of the primitive action, test, conditional and while loop constructs are extended with a $Knows(\varphi, s)$ predicate that explicitly requires the agent to know the value of the condition φ in order to proceed.

A compilation algorithm is presented that transforms the action theory with complex actions into a new theory where a complex action δ is replaced by a new primitive action $prim_\delta$. The new primitive action should be executable in s exactly when δ is executable in s. To ensure $prim_\delta$ preserves the physical effects of δ, for each fluent, effect axioms for $prim_\delta$ are added such that whenever δ makes F true/false, $prim_\delta$ will also make it true/false. Moreover, to preserve the knowledge effects of δ, $prim_\delta$ should emulate δ with respect to the K fluent. To write these new axioms, a special form of the regression operator R^s [112] is used to ensure that precondition and effect axioms only talk about situation s. For example, $Poss(prim_\delta(\vec{y}), s) \equiv R^s[(\exists s')Do(\delta(\vec{y}), s, s')]$ indicates that $prim_\delta$ can be executed in situation s if and only if δ can. The new theory is then used to find a plan that achieves a goal G. The authors show that to obtain a counterpart of this plan in the original theory, every occurrence of the $prim_\delta$ can be replaced by δ.

2.4.4 Norm Abstraction

Grossi and Digum [79, 78] use a KD45 multi modal logic corresponding to a propositional logic of contexts [108] to model norms at different levels of abstractions. Levels of abstraction constitute a structure ordered according to the relation "i is strictly less abstract than j". It is assumed that this relation is irreflexive, asymmetric, transitive and partial. The fact that holds in a level holds irrespective of the level from which that fact is considered. In addition, it is assumed that no inconsistency holds at any level. Finally, a trivial "outermost level" exists which represents the level of logical truths.

Translation rules connect the truth from more concrete to more abstract levels. For example, the notion of *counts as* is defined as follows: A *counts as* B if and only if A at level i determines the truth of B at a level j, where $i < j$, i.e., i is strictly less abstract than j. As different translation rules have contradictory consequents, the translation rules are assumed to be defeasbile. Deciding among conflicting defaults could be resolved in number of ways, for instance, by *specificity.*

In related work, Vázquez-Salceda and Dignum [162] introduce HARMONIA, a multi-level framework for modeling electronic organizations. This framework consists of four levels of abstraction, that can represent from the most abstract levels of normative systems to final implementation of rules and policies that can be used by agents.

Abstract norms are represented by the language *ANorms*, a denotic logic that is temporal, relativized and conditional (i.e., the obligation to reach a state or perform an action may be conditional on some state of affairs holding, moreover, it is meant for a certain agent role and should be fulfilled before a certain point in time).

To enable checking norms and acting on possible violations, the abstract norms need to be translated into *concrete norms*, which pertain to actions that are described in terms of the ontology of the organization. These norms are described in *CNorms*, which is the same language as *ANorms*, with the difference that it uses different predicates. The authors define a function $I : ANorms \rightarrow CNorms$ which is a mapping from abstract norms to concrete ones, and indicates how an abstract norm can be fulfilled by concrete norms. This function is based on the "counts as" operator [79, 78]. Several types of abstract norms and the concrete realizations are discussed; these include *Abstract actions* which can be implemented by different concrete actions, and *Temporal Abstractness* where an implicit deadline for obligations is made more concrete.

Translating concrete norms to rules also involves a change of language. The authors follow the approach of Mayer [114], that proposes a reduction from denotic logic to a Propositional Dynamic Logic. For example, a formula like $O(\alpha)$, which indicates α is obligatory, is reduced to dynamic logic as $O(\alpha) \equiv [\neg\alpha]V$ which expresses that α is obligatory if and only if not doing α leads to a violation. In this way, norms are translated into restrictions on behavior as well as triggers on unwanted behavior of agents interacting in the organization. While an organization can not force agents to do an action, it can prevent the agent from leaving before the action is done.

2.5 Supervisory Control of Discrete Event Systems

A Discrete Event System (DES) is a discrete-state, event-driven system, that is, its state evolution depends entirely on the occurrence of asynchronous discrete events over time [24]. An example of an event may correspond to the completion of a task, the failure of a machine in a manufacturing system or the arrival of a packet in a communication system. Such systems arise in a variety of areas including, computer operating systems, databases, communication networks and manufacturing systems. A number of models for DES have been proposed in the literature, such as automata [24], Petri nets [24] or process algebras [88]. The most common properties to be verified that arise in the study of software implementations of control systems for complex automated systems include safety (i.e., avoidance of illegal behavior), liveness (i.e., avoidance of deadlock and livelock), and diagnosis (i.e., ability to detect occurrences of unobservable events) [24].

Supervisory Control of Discrete Event Systems (SCDES) [129, 130, 166, 24], sometimes referred to as the Ramadge and Wonham (R&W) framework, provides a method for the synthesis of supervisors in discrete event systems that minimally constrain the behavior of a plant to ensure that a given specification is fulfilled. Ramadge and Wonham [130] represent a DES as an automaton with a set of states and a transition function. It is not assumed that the state set is finite, or that states have a specific structure. This allows for the possibility of counters and other infinite state devices. Here, we focus on finite-state representations of the plant and the supervisor.

Section 2.5.1 presents a brief overview of supervisory control theory for systems modeled by deterministic finite-state automata and subject to regular language specifications. Some extensions to the basic R&W framework are discussed in Section 2.5.2. Finally, in Section 2.5.4 we look at some of the main approaches in the area of reasoning about actions and change that are inspired by SCDES.

2.5.1 Controllability and Supervision

The plant G represents the "uncontrolled behavior" of the DES, whose behavior is assumed unsatisfactory and must be modified by feedback control in order to achieve a set of specifications. The *plant* G is modeled as a *generator*[10] which is a tuple of the form: $G = (Q, \Sigma, \delta, q_0, Q_m)$ where Q represents the finite set of states, Σ is a finite set of events associated with G and $\delta : Q \times \Sigma \to Q$ denotes a partial transition function. If a transition $\delta(q, \sigma)$ is defined, then it is denoted by $\delta(q, \sigma)!$. The initial state is represented by q_0 and $Q_m \subseteq Q$ is the set of *marked* states, used to mark the termination of certain event sequences, for example, those representing the completion of a task by the system. Let Σ^* denote the set of all finite strings of

[10]In the *accepting* mode, the automaton representing a regular language checks if an *input string* is part of the language. In the *generating* mode, the automaton *produces* the list of all the strings in the language. In general, only a proper subset of the totality of events can occur at each state of a generator. A generator may simply be a recognizer from which the dump state (if any) and all transitions to it have been dropped [166].

elements of Σ that contains the empty string ϵ. It is possible to extend δ to Σ^*, defined inductively as: $\delta(q,\epsilon) = q$ and $(\forall s \in \Sigma^*)(\forall \sigma \in \Sigma), \delta(q, s\sigma) = \delta(\delta(q,s),\sigma)$. Similarly, $(\forall s \in \Sigma^*), \delta(q,s)!$ if and only if $\exists q'$ such that $\delta(q,s) = q'$.

The behavior of the plant G is characterized by language $L(G)$, the prefix-closed language[11] generated by the plant, and the language $L_m(G)$ defined to be the set of all sequences of events which lead to a marked state, interpreted as completed tasks (or sequences of tasks) by the system that G models. More formally, the *closed behavior* of G is defined as $L(G) = \{s \in \Sigma^* \mid \delta(q_0, s)!\}$ and the *marked behavior* of G is defined as $L_m(G) = \{s \in L(G) \mid \delta(q_0, s) \in Q_m\}$. By definition, $L_m(G) \subseteq L(G)$.

To specify controllability of events, the R&W framework extends the basic automata by partitioning Σ into two disjoint subsets of *controllable* and *uncontrollable* events: $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$. The set of controllable events, Σ_c, are the events that can be prevented from happening, or disabled, by a controller; Σ_u represents the set of uncontrollable events that remain enabled all the time. There are many reasons why an event would be modeled as uncontrollable; for instance it is inherently unpreventable (e.g., a fault event), it models a change of sensor readings not due to a command, or it is modeled as uncontrollable by choice, e.g., when the event has high priority and thus should not be disabled [24].

Modifying the behavior of G is understood as restricting the behavior to a subset of $L(G)$ which represent the "legal" or "admissible" behavior of G. The restriction of system behavior is enforced by a *supervisor*, that observes the events executed by G, and then disables some of the controllable events in order to satisfy the specification. Formally, a supervisor is a function $S : L(G) \to \Gamma$ where $\Gamma = \{\gamma \in 2^\Sigma \mid \gamma \supseteq \Sigma_u\}$ that maps from the sequence of events generated so far to a set of "enabled" events that contains the uncontrollable events in addition to a subset of controllable events that G can execute in its current state. G under the supervision of S is denoted by S/G. The closed behavior generated by S/G is defined as the language $L(S/G) \subseteq L(G)$ and described as the least set such that:

1. $\epsilon \in L(S/G)$
2. if $[s \in L(S/G), \sigma \in S(s)$ and $s\sigma \in L(G)]$ then $[s\sigma \in L(S/G)]$

$L(S/G)$ is prefix-closed by definition. $S(s)$ refers to the control action at s. The empty string ϵ is always in $L(S/G)$ since it is always contained in $L(G)$. When S is adjoined to G, then G is to start in its initial state at which time its possible first transition will be constrained by the control action $S(\epsilon)$. The marked behavior of S/G is $L_m(S/G) = L(S/G) \cap L_m(G)$. Thus the marked behavior of S/G consists exactly of the

[11] The *prefix-closure* $\overline{L}$ of language L is defined to be the set of all prefixes of strings in the language; more formally: if $L \subseteq \Sigma^*$ then $\overline{L} = \{s \in \Sigma^* \mid (\exists t \in \Sigma^*)\ st \in L\}$. If $L = \overline{L}$, then L is called a *closed (or prefix-closed)* language.

strings of $L_m(G)$ that "survive" under supervision by S. The supervisor S is said to be *non-blocking* (for G) if $\overline{L_m(S/G)} = L(S/G)$. Figure 2.1 shows the closed loop between the supervisor and the plant.

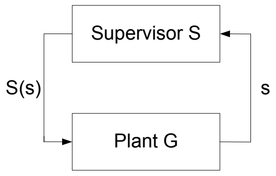

Figure 2.1: The Feedback Loop of Supervisory Control

Given a plant G, two questions that arise are 1) what sublanguages of $L(G)$ qualify as $L(S/G)$ for some S? and 2) what are the sublanguages of $L_m(G)$ that qualify as $L_m(S/G)$ for some non-blocking S for G? A key notion that enables answering these (and more) questions is the concept of *controllability*. A language $K \subseteq \Sigma^*$ is called *controllable* with respect to a system G if

$$\forall(s,\sigma)\ s \in \overline{K}\ \&\ \sigma \in \Sigma_u\ \&\ s\sigma \in L(G) \Rightarrow s\sigma \in \overline{K} \quad \text{or more concisely,} \quad \overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}$$

In other words, if something cannot be prevented, it must be legal. It is clear that the empty language $\emptyset$, $L(G)$ and Σ^* are always controllable (with respect to G).

Given a plant G, with closed behavior $L(G)$ and marked behavior $L_m(G)$, two important results of the R&W framework are as follows: 1) Let $K \subseteq L(G)$ be nonempty and closed. There exists a supervisory control S for G such that $L(S/G) = K$ if and only if K is controllable with respect to G; 2) Let $K \subseteq L_m(G)$, $K \neq \emptyset$. There exists a non-blocking supervisory control (NSC) S for G such that $L_m(S/G) = K$ if and only if K is controllable with respect to G and K is $L_m(G)$-closed [166].[12]

A slight generalization of the NSC, in which the supervisory action includes *marking*, as well as control, is defined as follows. Let $M \subseteq L_m(G)$. A *Marking Non-blocking Supervisory Control* (MNSC) for the pair (M, G), is a map $S : L(G) \to \Gamma$, defined exactly as above, with exception of the marked behavior of S/G which is defined as $L_m(S/G) = L(S/G) \cap M$. Let $K \subseteq L_m(G)$, $K \neq \emptyset$. There exists a MNSC S for (K, G) such that $L_m(S/G) = K$ if and only if K is controllable w.r.t G [166].

The above cases assume that the specification language is controllable with respect to G. But what if a given specification language $E \subseteq \Sigma^*$ is not controllable (with respect to G)? Then it is desirable to find a controllable approximation to the given language, preferably the largest controllable language contained in

[12]A language $L1$, where $L1 \subseteq L2 \subseteq \Sigma^*$, is said to be *L2-closed*, if $L1 = \overline{L1} \cap L2$. Thus $L1$ is $L2$-closed provided it contains every one of its prefixes that belongs to $L2$.

E. This language should preserve the restrictions imposed by E, while requiring the least amount of control. It can thus be regarded as the "optimal" or "minimally restrictive" approximation to E.

For a given $E \subseteq \Sigma^*$, it is always possible to define the set of its controllable sublanguages (with respect to G), described as $C(E) = \{K \subseteq E \mid \text{K is controllable with respect to G}\}$. $C(E)$ is nonempty because the empty language ($\emptyset$) is controllable, hence always belongs to this set. Moreover, $C(E)$ is partially ordered by inclusion and closed under arbitrary unions. In particular, it contains a unique supremal element, called $supC(E) = \cup\{K \mid K \in C(E)\}$ [166, 130]. $supC(E)$ is sometimes denoted as $E^{\uparrow C}$. When E and $L(G)$ are regular languages, then so is $E^{\uparrow C}$ [24].

Let $E \subseteq \Sigma^*$ be $L_m(G)$-marked[13], and let $K = supC(E \cap L_m(G))$. If $K \neq \emptyset$, there exists a non-blocking supervisory control (NSC) S for G such that $L_m(S/G) = K$. Moreover, let $E \subseteq \Sigma^*$ and let $K = supC(E \cap L_m(G))$. If $K \neq \emptyset$ there exists a marking non-blocking supervisory control (MNSC) S for (K,G) such that $L_m(S/G) = K$ [166].

The abstract definition of a supervisory control as a map $L(G) \to \Gamma$ does not in itself provide a concrete representation for practical implementation. As a first step, it is possible to construct a trim automaton $EDES$ that represents the desired behavior E of the plant, such that $L_m(EDES) = E$ and $L(EDES) = \overline{E}$.

Given a specification language E ($E = L_m(EDES)$ as defined above), an algorithm for the computation of the supremal controllable sublanguage $K = supC(E \cap L_m(G)) = E^{\uparrow C}$ is presented by Wonham [166]. Let $Pwr(\Sigma^*)$ denote the power set of Σ^*. A language operator, $\Omega : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$, is defined according to:

$$\Omega(Z) = E \cap L_m(G) \cap sup\{T \subseteq \Sigma^* \mid T = \overline{T}, T\Sigma_u \cap L(G) \subseteq \overline{Z}\}$$

It is shown that K is the largest fixpoint of Ω. In case of regular languages, this fixpoint can be computed by successive approximation. Let $K_0 = E \cap L_m(G)$, $K_{j+1} = \Omega(K_j)$ where $(j = 1, 2, 3, ...)$. It can be shown that $K = \lim K_j$ $(j \to \infty)$. If G and $EDES$ have m and n states respectively, then this scheme converges after at most mn iterations. As the computation of Ω is itself bounded by a polynomial in m and n, it means that the computation of the supremal controllable sublanguage is of polynomial complexity in m and n [130].

In supervisory control, while disabled events are certainly prevented from occurring, enabled events are not necessarily "forced" to occur. Also, the supervisor exerts "dynamic" feedback control on G, in the sense

[13]With $L1, L2 \subseteq \Sigma^*$, $L1$ is said to be $L2$-*marked* if $L1 \supseteq \overline{L1} \cap L2$, namely any prefix of $L1$ that belongs to $L2$ must also belong to $L1$, but $L1$ may have more elements. Two languages $L1, L2 \subseteq \Sigma^*$ are *non-conflicting* if $\overline{L1 \cap L2} = \overline{L1} \cap \overline{L2}$.

that the decision about which events to disable may change whenever S observes the execution of a new event by G.

2.5.2 Extensions to Supervisory Control of Discrete Event Systems

The basic R&W framework has been extended in number of directions. For example, non-determinism in a discrete event system is often considered as the result of lack of information (e.g., partial observation or unmodeled internal dynamics of a system). Non-deterministic plants e.g., [92], non-deterministic supervisors e.g., [87] and non-deterministic specifications e.g., [174] have been studied.

When the plant is very complex or time-varying,[14] it may not be possible to model the plant a priori due to the large set of states or the lack of information on the possible system variations. In such cases, after the occurrence of each event, the online *limited-lookahead controller* [25, 76, 159] uses a N-Step ahead projection (a tree) of all the possible continuations of the executed prefix to make a control decision.

Another direction, mainly motivated by improving the understandability of the control logic and reducing the computational effort of the synthesis of a monolithic supervisor in complex and/or distributed systems has resulted in more modular and hierarchical architectures. Such models focus on architectural decomposition of the control structure, while aiming to achieve global optimality (maximal permissiveness) and non-blockingness. Examples include decentralized, distributed, hierarchical and heterarchical architectures. Decentralized supervisory control [134, 133] is concerned with systems where several “site supervisors” work as a team to control a system that is inherently distributed, such as computer networks. Supervisors at each site may see the effect of different (possibly overlapping) sets of sensors and may control different (possibly overlapping) sets of actuators. Distributed architectures [23, 171] aim to allocate external supervisory control action to individual plant components, as their internal control strategies. Hierarchical approaches to supervision [173, 165, 123] break down the complexity in a vertical fashion, whereby controllers may operate on different levels of logical or temporal abstraction. Finally, heterarchical architectures [96, 146, 148] are inspired from hierarchical and decentralized approaches.

2.5.3 Hierarchical Supervisory Control of Discrete Event Systems

Hierarchical supervisory control of DES involves modeling a plant and its supervisor at an abstract level, and using the control decisions of the abstract supervisor to guide the decisions of a supervisor of the original plant. The basic model of hierarchical supervision initially proposed by Zhong and Wonham [172] has been extended in a number of directions, for example, by incorporating partial observability [20] and hierarchical control of decentralized plants [147].

[14]Dynamic Discrete Event Systems (DDES) [76, 77] are a class of time-varying systems composed of DES modules, where at each time period, the system may consist of a composition of different set of modules.

The hierarchical control of DES proposed by Zhong and Wonham [172, 166] introduces a two-level hierarchy consisting of a low-level plant G_{lo} and controller C_{lo}, along with a high-level plant G_{hi} and controller C_{hi}. These are coupled as illustrated in Figure 2.2.

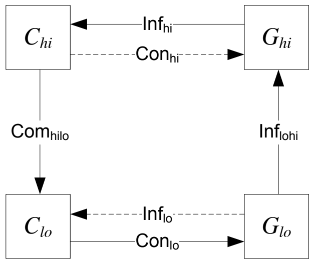

Figure 2.2: Two-Level Control Hierarchy [166]

$G_{lo} = (Q, \Sigma, \delta, q_0, Q_m)$ is the actual plant to be controlled in the real world by the "operator" C_{lo}. The initial assumption is that $Q_m = Q$, therefore the relevant languages are prefix-closed.[15] G_{hi} is an abstract and simplified model of G_{lo} used for decision-making by the "manager" C_{hi}. Control channels are represented by Con_{hi} and Con_{lo} and the information feedback channels are modeled by Inf_{hi} and Inf_{lo}. G_{lo} uses the bottom-up information channel Inf_{lohi} to "report" events to G_{hi}. More formally, given T, the event set of the high-level system, the information channel (or abstraction mapping) Inf_{lohi} is modeled by using a causal reporter map, $\theta : L(G_{lo}) \to T^*$, where $s \in L(G_{lo})$ and $\sigma \in \Sigma$, such that:

$$\theta(\epsilon) = \epsilon \ ,$$

$$\theta(s\sigma) = \begin{cases} \text{either } \theta(s) \\ \text{or } \theta(s)\tau, \text{some } \tau \in T \end{cases}$$

Informally, θ can be used to signal events that depend on the past history of the behavior of G_{lo}; for example, θ might produce an instance of symbol τ' whenever G_{lo} generates multiple of 5 of some distinguished symbol σ'. The causal reporter map θ is interpreted as "summarizing" every sequence of low-level events in the form of a high-level event sequence (which is typically shorter than the low-level one). In this way, G_{hi} generates a language equal to the image of the low-level language under θ, i.e., $L(G_{hi}) = \theta(L(G_{lo}))$. Hence, the behavior of the high-level system is *driven* by the behavior of the low-level system.

In fact, it is possible to represent θ with G_{lo} by a *Moore* generator[16] having the output alphabet $T_o = T \cup \{\tau_o\}$, where $\tau_o \notin T$ is interpreted as the "silent output symbol". The states of G_{lo} that output events

[15]This assumption will be generalized towards end of this section to include marking and non-blockingness.

[16]Moore automata are a type of automata with (state) outputs. There is an output function that assigns an output to each state, and this output is "emitted" by the automaton when it enters a state [24].

other than τ_o are referred to as "vocalized". The following defines an output map from $s \in L(G_{lo})$ to a state output: $\hat{\omega} : L(G_{lo}) \rightarrow T_o$ such that $\hat{\omega}(\epsilon) = \tau_o$, and if $s\sigma \in L(G_{lo})$ then,

$$\hat{\omega}(s\sigma) = \begin{cases} \tau_o \text{ if } \theta(s\sigma) = \theta(s) \\ \tau \text{ if } \theta(s\sigma) = \theta(s)\tau \end{cases}$$

In order to equip G_{hi} with a control structure, Zhong and Wonham [173] propose a technique that based on the controllability of individual low-level events that form a sequence of events summarized by a high-level event, partitions the high-level alphabet T into controllable (T_c) and uncontrollable (T_u) subsets. G_{lo} is considered to be *output-control-consistent* (OCC) when, for all $\tau \in T$, it is unambiguous if τ is controllable or uncontrollable. τ is considered uncontrollable if all the sequence of events in its refinements are uncontrollable; otherwise it is considered controllable. If G_{lo} is not OCC, an algorithm exists that refines descriptions of G_{hi} and G_{lo}, and extends T to partition it into controllable and uncontrollable events.

The high-level supervision is only virtual. In other words, C_{hi} has no direct influence over G_{hi}. Based on a high-level specification, the appropriate control decisions of C_{hi} are communicated via the top-down command channel Com_{hilo} to the low-level supervisor C_{lo} which in turn, interprets them as appropriate control to be applied on G_{lo}.

More formally, the high-level supervisory control is determined by a selection of high-level controllable events to be disabled, on the basis of high-level past history; thus C_{hi} is defined by a map $\gamma_{hi} : L(G_{hi}) \times T \rightarrow \{0, 1\}$ such that $\gamma_{hi}(t, \tau) = 1$ for all $t \in L(G_{hi})$ and $\tau \in T_u$, i.e., all uncontrollable events are enabled, while some controllable events $\tau \in T_c$ may be disabled. Given γ_{hi}, the *high-level disabled-event map* is defined as $\Delta_{hi} : L(G_{hi}) \rightarrow 2^{T_c}$ according to: $\Delta_{hi}(t) = \{\tau \in T_c \mid \gamma_{hi}(t, \tau) = 0\}$. In turn, the corresponding *low-level disabled-event map* is described as $\Delta_{lo} : L(G_{lo}) \times L(G_{hi}) \rightarrow 2^{\Sigma_c}$, according to

$$\Delta_{lo}(s, t) = \{\sigma \in \Sigma_c \mid (\exists s' \in \Sigma_u^*) s\sigma s' \in L(G_{lo}) \ \& \ \hat{\omega}(s\sigma s') \in \Delta_{hi}(t) \ \& \ (\forall s'') s'' < s' \Rightarrow \hat{\omega}(s\sigma s'') = \tau_o\}$$

In the above, $s'' < s'$ refers to string s'' as a prefix of string s'. Informally, the low-level disabled map designates those controllable events to be disabled that form the last controllable event in a series of events that are summarized by each high-level event which is included in the high-level disabled map (thus achieving maximal permissiveness). Since the hierarchical loop is closed through Inf_{lohi}, a string $s \in L(G_{lo})$ is mapped to $t = \theta(s) \in L(G_{hi})$. Then the control implemented by C_{lo}, namely $\gamma_{lo}(s, \sigma)$ is to disable the event if it is in the low-level disabled event map and allow it otherwise.

Given a non-empty, closed specification language $E_{hi} \subseteq L(G_{hi})$, which is controllable with respect to the high-level model structure, the (maximal) behavior in G_{lo} is defined to be $E_{lo} = \theta^{-1}(E_{hi}) \subseteq L(G_{lo})$. Clearly,

$\theta(E_{lo}) = E_{hi}$ (since $L(G_{hi}) = \theta(L(G_{lo}))$), and moreover, E_{lo} is closed. Assume $L(\gamma_{lo}, G_{lo})$ represents the closed-loop language synthesized in G_{lo}. A major result presented by Zhong and Wonham [173] states that $L(\gamma_{lo}, G_{lo}) = E_{lo}^{\uparrow C}$. This result is referred to as *low-level hierarchical consistency.* Informally, it ensures that the updated behavior of G_{hi} will always satisfy the high-level legal constraint $\theta(L(\gamma_{lo}, G_{lo})) \subseteq E_{hi}$, and that the "real" low-level behavior in G_{lo} will be as large as possible according to this constraint.

In some cases, only a subset of the "expected" high-level specification may be achievable (hence the inclusion in $\theta((\theta^{-1}(E_{hi}))^{\uparrow C}) \subseteq E_{hi}$ will be proper). The reason is that a call by C_{hi} for the disablement of some high-level event $\tau \in T_c$ may require C_{lo} (i.e., the control γ_{lo}) to disable paths in G_{lo} that lead directly to outputs other than τ. If $(\theta((\theta^{-1}(E_{hi}))^{\uparrow C}) = E_{hi})$, for every closed and controllable language $E_{hi} \subseteq L(G_{hi})$, then the pair (G_{lo}, G_{hi}) is *hierarchically consistent.* A sufficient condition for hierarchical consistency is that of *strict output-control consistency*, which simply means that the system is output-control-consistent and the low-level system allows for the enabling and disabling of every controllable high-level event independently.

Two results by Wong and Wonham [165] place the property of hierarchical consistency in clear perspective. Given a G_{lo} that is OCC, it is possible to define the *Main Condition*(MC) as: $\theta C(L(G_{lo})) = C(L(G_{hi}))$, where $C(L(G_{lo}))$ and $C(L(G_{hi}))$ represent the family of all controllable sublanguages of $L(G_{lo})$ and $L(G_{hi})$ respectively. MC states that not only θ preserves controllability, but also that every high-level controllable language is the θ-image of some (possibly more than one) low-level controllable language. This amounts to equating executable tasks with controllable languages: every task that could be specified in the manager's abstracted (aggregated) model G_{hi} is executable in the operator's (detailed) model G_{lo}; thus, high-level policies can always be carried out operationally.

Assume $E_{hi} \subseteq L(G_{hi})$, a high-level legal specification (which may not be controllable), is proposed to the operator by specification $\theta^{-1}(E_{hi})$. The operator may then synthesize $(\theta^{-1}(E_{hi}))^{\uparrow C} \subseteq L(G_{lo})$, which results in $(\theta(\theta^{-1}(E_{hi}))^{\uparrow C})$ being implemented in G_{hi}. It is desired that this implemented sublanguage of $L(G_{hi})$ to be the language $E_{hi}^{\uparrow C}$ that a manager would synthesize directly (if direct control were viable). This forms the essence of hierarchical consistency, and the following result states that hierarchical consistency in this strong sense is equivalent to MC.

$$MC \Leftrightarrow [(\forall E_{hi}) E_{hi} \subseteq L(G_{hi}) \Rightarrow \theta((\theta^{-1}(E_{hi}))^{\uparrow C}) = E_{hi}^{\uparrow C}]$$

If MC is slightly weakened, then the following related result can be proved, as a simpler version of the condition of hierarchical consistency:

$$\theta C(L(G_{lo})) \supseteq C(L(G_{hi})) \Leftrightarrow [(\forall E_{hi}) E_{hi} \in C(L(G_{hi})) \Rightarrow \theta((\theta^{-1}(E_{hi}))^{\uparrow C}) = E_{hi}]$$

Informally, the fact that the family of all controllable sublanguages of $L(G_{hi})$ is a subset of the θ image of the family of all controllable sublanguages of $L(G_{lo})$ is equivalent to the fact that for every controllable high-level legal specification E_{hi}, the θ image of the supremal controllable sublanguage obtained from $\theta^{-1}(E_{hi})$ in the low-level, implements E_{hi} in the high-level plant.

The theory of hierarchical supervision has also been extended to include marking and non-blocking [166]. Let $L_m(G_{lo}) \subseteq L(G_{lo})$ and $L_m(G_{hi}) = \theta(L_m(G_{lo}))$ represent the marked behaviors of G_{lo} and G_{hi} respectively. Given G_{lo} that is OCC, and a specification language $E_{hi} \subseteq L_m(G_{hi})$ for G_{hi}, the marked behavior "virtually" synthesized in G_{hi} is $K_{hi} = \mathit{supC}(E_{hi})$. The specification which is "announced" to the low-level controller is $\theta^{-1}(E_{hi})$. In this way, the marked behavior synthesized in G_{lo} is $K_{lo} = \mathit{supC}(L_m(G_{lo}) \cap \theta^{-1}(E_{hi}))$. The desired property that should be satisfied is: $\theta(K_{lo}) = K_{hi}$. More formally, it is shown that

$$(\forall E_{hi}) E_{hi} \subseteq L_m(G_{hi}) \Rightarrow \theta(K_{lo}) = K_{hi} \text{ (HCm)} \quad \text{iff} \quad \theta C(L_m(G_{lo})) = C(L_m(G_{hi})) \text{ (MCm)}$$

To satisfy main condition with marking (MCm), the causal reporter map θ is enhanced with a *global observer* property; moreover, a type of local controllability in G_{lo} is ensured.

Assume $L_{voc} = \{s \in L(G_{lo}) \mid s = \epsilon \text{ or } s \text{ ends with a vocal state}\}$. Extending the map with a global observer property informally means that whenever $\theta(s)$ can be extended to a string $t \in L(G_{hi})$, the underlying L_{voc} string s can be extended (by string s') to L_{voc} string ss' with the same image under θ. In other words, "the manager's expectation can always be executed in G_{lo}", at least when starting from a L_{voc} string.

To implement the local controllability in G_{lo}, first it is assumed that the control decisions are delegated to "agents", e.g., $Agent(s)$ for each $s \in L_{voc}$. The scope of $Agent(s)$ is the local language $L_{voc}(s)$ which links the vocal node at s to adjacent downstream vocal nodes (if any) in the reachability tree of L. $L_{lo}(s)$ is defined as the prefix closure of $L_{voc}(s)$. In addition, the local reporter map $\theta_s : L_{lo}(s) \rightarrow \{\epsilon\} \cup T$ is defined that maps each s' to τ, if $\hat{\omega}(ss') = \tau$; otherwise s' is mapped to ϵ. Furthermore, $C_{lo}(s)$ is defined as the family of all controllable sublanguages of $L_{lo}(s)$.

The authors define $L_{hi}(t)$ as (at most) one-step closed sublanguage that is postfix to $t = \theta(s)$, as well as $C_{hi}(t)$ as the controllable sublanguages of $L_{hi}(t)$. The local controllability property of interest is that $C_{hi}(t) = \theta_s(C_{lo}(s))$. When this property holds it is said that G_{lo} is *locally output controllable at* s; and if it holds for all $s \in L_{voc}$, it is said that G_{lo} is *globally output controllable*. It is shown that if θ is an L_{voc} observer, G_{lo} is locally output controllable, and $L_m = \theta^{-1}(M_m) \cap L_{voc}$ (i.e., the marked state of the low-level system are only those vocalized states that map back to the marked states of the high level system), then the main condition with marking is satisfied ($\theta C(L_m(G_{lo})) = C(L_m(G_{hi}))$).

2.5.4 Reasoning About Actions and Change and Supervisory Control of Discrete Event Systems

DeGiacomo et al. [35], inspired by the supervisory control of discrete event systems, proposed *agent supervision*, which is a framework of control and customization of agent behavior. This framework is based on the situation calculus and ConGolog agent programming language. We provide a more detailed discussion of this work in Section 3.3.

Aucher [7] reformulates some of the results of supervisory control in the areas of partial controllability, partial observability and decentralized control in terms of model checking problems expressed in an epistemic temporal logic. The epistemic temporal logic considered is $CTL^*K_n^D$ [161], which has an asynchronous semantics with perfect recall[17] and branching time. An "environment" transition system is introduced that is the combination of the transition system of the plant G and the transition system O representing the specification language K. Then an interpreted system (I) associated to G and O is defined. Different conditions for fully realizing a goal behavior under partial controllability, partial observation or decentralization of supervisors are formulated in $CTL^*K_n^D$. Moreover, it is shown that in case of control with full realization, for any supervisor S, any $w \in \mathrm{Ł}(G)$, and any $\sigma \in \Sigma_c$, there is a formula $F(\sigma)$ of $L_{CTL^*K_n^D}$ such that $\sigma \in S(w)$ if and only if $I, w \models F(\sigma)$. Representing supervisors in terms of model checking problems allows lazily computing them online. This approach is only applicable on finite-state systems.

Felli et al. [58] relate the notion of a composition controller in the "Roman Model" approach to behavior composition to that of a supervisor in SCDES. Given an available system of behaviors $S = \langle B_1, \ldots, B_n \rangle$ which is formed by joint execution of available (non-deterministic) behaviors $B_i, i \in \{1, \ldots, n\}$, and a desired deterministic target behavior T, first a plant $G_{S,T}$ is built from S and T. Controlling $G_{S,T}$ consists of controlling behavior delegations. A supervisor can enable or disable an available behavior to execute; however, the supervisor cannot control the action requests or the evolution of the behavior selected (they are modeled as uncontrollable events). A state in $G_{S,T}$ encodes a snapshot of the composition process: the state of all available behaviors and the target as well as the current pending target request and current behavior delegation. The states with no pending request or delegation are considered marked (final). The plant transition function is defined in a way that the complete process for one target request and action delegation involves three transitions in the plant: *i*) target action request, *ii*) behavior delegation, and *iii*) available system evolution. In the initial state, and after each target request has been fulfilled, the plant is in a state where no active request or no behavior delegation exist, and ready to process a new target request

[17]The asynchronous perfect recall semantics assumes that the system operates asynchronously, with agents aware of the passing of time only when their observations change. However, agents remember the sequence of distinct observations they have made [161].

(i.e., a marked state). In case the chosen behavior is unable to execute the delegation from its current state, then no transition is defined and the plant (non-marked) state is a dead-end.

The specification language $K_{S,T}$ is then defined as exactly the marked language of the composition plant, that is $L_m(G_{S,T})$. In other words, the plant is controlled in a way so as to eventually be able to reach the end of each request-delegation process. It is shown that the supervisors able to control the specification $K_{S,T}$ in plant $G_{S,T}$ correspond one-to-one with the composition (solution) controllers for building target T in available system S. The authors then propose a technique for extracting the controller generator (i.e., an implicit representation of all controllers) from the supervisor of the plant. This technique is based on compressing the transitions in the generator which represents the behavior of the controlled system.

When no exact composition for a target T in a system S exists, the authors suggest adapting the composition plant $G_{S,T}$ to look for Supremal Realizable Target Fragments (SRTFs) [169] for the special case of deterministic available behaviors. An SRTF models a fragment of the target behavior that accommodates an exact composition and is closets to the target module. Since it is not possible to realize all the target traces, user's requests and delegations are modeled as controllable events. In this way, the supervisor can enable or disable requests to ensure as many as possible of the target traces are realized.

3 Foundations

This chapter provides the necessary background and the theoretical foundation for this dissertation. In Section 3.1 we provide an overview of Reiter's version of the situation calculus [132]. The next section focuses on ConGolog, one of the high-level programing languages defined over the situation calculus. Finally, in Section 3.3, we present the agent supervision framework proposed by DLM [35].

3.1 The Situation Calculus

3.1.1 The Language

The *Situation Calculus* [110, 132] is a predicate logic language designed for representing and reasoning about dynamically changing worlds. In this dissertation, we focus on Reiter's version of situation calculus [132], which is a sorted dialect of first order logic with equality (with some second order elements) $L_{Sitcalc}$. It includes three disjoint sorts: **action** for *actions*, **situation** for *situations* and a catch-all sort **object** for everything else depending on the domain of application.

Actions are assumed to be the cause of all changes in the world; they are represented as terms in the logic. Function symbols of sort $object \mapsto action$ are used for terms $A(\vec{x})$ which represent *action types.* For instance, $deliver(Shipment1, Warehouse1)$ could stand for the action of delivering the shipment $Shipment1$ to the warehouse $Warehouse1$. They are referred to as actions types since a single function symbol may be instantiated with different parameters; for example $deliver(Shipment2, Warehouse2)$. The lower case letters a_1, a_2, ... denote action variables, and α_1, α_2, ... denote action terms. In this dissertation, we assume there is a finite number of action types $\mathcal{A}$.

A *situation* is a term in $L_{Sitcalc}$ that represents a possible world history of the primitive actions performed so far. The constant S_0 denotes the initial situation where no actions have been performed yet. There is a special function symbol do of sort $(action, situation) \mapsto situation$; $do(a, s)$ represents the successor situation resulting from performing action a in situation s. For instance, $do(a_3, do(a_2, do(a_1, S_0)))$ is a situation term denoting the sequence of actions $[a_1, a_2, a_3]$. We write $do([a_1, a_2, \ldots, a_{n-1}, a_n], s)$ as an abbreviation for the situation term $do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s))\ldots))$; for an action sequence $\vec{a}$, we often write $do(\vec{a}, s)$ for $do([\vec{a}], s)$. The binary relation $\sqsubset$ is used to define precedence on situations; thus $s_1 \sqsubset s_2$ indicates s_1 is a sub-history of s_2.

Fluents are one of the main language elements of the situation calculus. These are predicates or functions that are used to describe what holds in a situation; their value may vary from situation to situation as a result of execution of actions based on domain-specific causal laws. By convention, the last argument of a fluent is a situation. For example, $At(Shipment1, Warehouse1, s)$ denotes that $Shipment1$ is at $Warehouse1$ in situation s. A *situation suppressed* fluent (resp. formula) is a fluent (resp. formula) where all occurrences of situation terms are deleted. If ϕ is a situation suppressed formula then $\phi[s]$ denotes the situation calculus formula with the suppressed situation s restored.

3.1.2 Basic Action Theories

A basic action theory (BAT) is a logical theory in the situation calculus that includes descriptions of what holds initially in the world, when actions can be performed, and how the world evolves under the effects of actions. More formally, a BAT $\mathcal{D}$ is the union of the following disjoint sets:

$$\mathcal{D} = \mathcal{D}_{poss} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \Sigma \tag{3.1}$$

Action Precondition Axioms $\mathcal{D}_{poss}$ Actions typically have preconditions, that is, conditions that need to hold for the action to occur. In the situation calculus, a special predicate $Poss(a, s)$ is used to state that action a is executable in situation s. For each action $A(\vec{x})$, the axiomatizer provides an axiom of the form (here and in the rest all free variables are assumed to be universally quantified from the outside):

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s) \tag{3.2}$$

where $\Pi_A(\vec{x}, s)$ is a first order formula *uniform* in s with free variables among $\vec{x}, s$. A formula of $L_{Sitcalc}$ is *uniform* in s if and only if it does not mention the predicates $Poss$ or $\sqsubset$, it does not quantify over variables of sort situation, it does not mention equality on situations, and whenever it mentions a term of sort situation in the situation argument position of a fluent, then that term is s. This results in the truth value of the formula depending only on situation s. Consequently, whether $A(\vec{x})$ can be performed in a situation s depends entirely on s. The abbreviation $Executable(s)$ is used to denote that every action performed in reaching situation s was possible in the situation in which it occurred.

As an example, suppose we have an action $deliver(sID)$ that can be performed to deliver a shipment with ID sID at its destination. This action has the following precondition axiom: $Poss(deliver(sID), s) \equiv \exists l.Dest(sID, l) \wedge At(sID, l, s)$, where the non-fluent predicate $Dest(sID, l)$ specifies the destination of the shipment (location l) and the fluent $At(sID, l, s)$ indicates the location of the shipment in situation s. Thus, action *deliver*(sID) is executable in situation s, if and only if, the destination of the shipment is location l, and the shipment is at location l in situation s.

Successor State Axioms $\mathcal{D}_{ssa}$ Actions typically also have effects, that is, fluents that are changed as a result of executing the action. In a BAT, for each predicate fluent F, a successor state axiom is defined in a way that provides a solution to the frame problem; it has the following form:

$$F(\vec{x}, do(a,s)) \equiv \Pi_F^+(\vec{x}, a, s) \vee [F(\vec{x}, s) \wedge \neg\Pi_F^-(\vec{x}, a, s)] \tag{3.3}$$

where $\Pi_F^+(\vec{x}, a, s) \vee [F(\vec{x}, s) \wedge \neg\Pi_F^-(\vec{x}, a, s)]$ is a formula uniform in s, all of whose free variables are among a, s, $\vec{x}$. $\Pi_F^+(\vec{x}, a, s)$ specifies all the conditions under which $F(\vec{x})$ becomes true when a is performed in s and $\Pi_F^-(\vec{x}, a, s)$ specifies all the conditions under which the fluent becomes false. It is assumed that action effects are deterministic, i.e., $\neg\exists\vec{x}, a, s.\Pi_F^+(\vec{x}, a, s) \wedge \Pi_F^-(\vec{x}, a, s)$, thus, no action a satisfies the condition of making the fluent F both true and false. It is further assumed that distinct actions have distinct names/arguments (see axioms 3.6 and 3.7 below). Formula 3.3 completely characterizes the value of fluent F in the successor state resulting from performing action a in situation s. Specifically, F is true after doing a if and only if before doing a, Π_F^+ was true, or both F and $\neg\Pi_F^-$ were true.

For instance, suppose the successor state axiom for the fluent $At(sID, l, do(a,s))$ is defined as

$$\begin{aligned} At(sID, l, do(a,s)) \equiv\ & \exists l'.a = moveShipment(sID, l', l) \vee \\ & At(sID, l, s)\ \wedge \forall l'.a \neq moveShipment(sID, l, l') \end{aligned}$$

Thus, shipment sID is at location l after performing action a if and only if, either action a was the *moveShipment* action which moved the shipment from a previous location l' to location l, or, the shipment was already at location l in situation s, and no *moveShipment* action was performed that would move the shipment to a new location l'.

A similar solution is provided for functional fluents. Given a functional fluent f, let $\Pi_F(\vec{x}, y, a, s)$ be a formula that characterizes all the conditions under which action a can cause f to change its value and take the value y in situation $do(a,s)$, that is:

$$f(\vec{x}, do(a,s)) = y \equiv \Pi_F(\vec{x}, y, a, s) \vee [y = f(\vec{x}, s) \wedge \neg\exists y'\Pi_F(\vec{x}, y', a, s)] \tag{3.4}$$

where $\Pi_F(\vec{x}, y, a, s) \vee [y = f(\vec{x}, s) \wedge \neg\exists y'\Pi_F(\vec{x}, y', a, s)]$ is a formula uniform in s, all of whose free variables are among $\vec{x}$, y, a, s. The above axiom states that fluent f has value y in situation $do(a,s)$ if and only if either action a caused f to take value y, or f had already value y in situation s and action a did not change its value. It is assumed that f satisfies the following consistency property:

$$\neg(\vec{x}, y, y', a, s).\Pi_F(\vec{x}, y, a, s) \wedge \Pi_F(\vec{x}, y', a, s) \wedge y \neq y' \tag{3.5}$$

The above consistency property states that action a cannot cause f to take two different values.

Initial State Axioms $\mathcal{D}_{S_0}$ Initial state axioms describe the initial state of the world (i.e., the one the agent starts with, before any actions have been executed). $\mathcal{D}_{S_0}$ is a set of first-order sentences uniform in S_0. Therefore, no sentence of $\mathcal{D}_{S_0}$ quantifies over situations, mentions Poss, $\sqsubset$ or the function symbol *do*. $\mathcal{D}_{S_0}$ may also contain sentences mentioning no situation term at all. Note that $\mathcal{D}_{S_0}$ may be an incomplete specification and may have many models. For example, suppose that we have the following initial state axioms:

$$Warehouse1 \neq Warehouse2,$$
$$\forall sID, w, w'.At(sID, w, S_0) \wedge At(sID, w', S_0) \supset w = w',$$
$$At(123, Warehouse1, S_0) \vee At(123, Warehouse2, S_0)$$

Thus, we have two distinct warehouses $Warehouse1$ and $Warehouse2$. Any shipment with ID *sID* can only be located at a single location (i.e., warehouse) in the initial situation. Moreover, shipment 123 may be at $Warehouse1$ or $Warehouse2$ in the initial state. In some models of the theory, $At(123, Warehouse1, S_0)$ holds, while in other models $At(123, Warehouse2, S_0)$ is true.

Unique Name Axioms for Actions $\mathcal{D}_{una}$ These axioms ensure every action function is one-to-one; the only action terms that can be equal are two identical actions with equal arguments.

For any distinct action names A and B,

$$A(\vec{x}) \neq B(\vec{y}) \tag{3.6}$$

Moreover, equal actions have equal arguments:

$$A(\vec{x}) = A(\vec{y}) \supset (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \tag{3.7}$$

Foundational Axioms Σ The following four *foundational axioms*, denoted by Σ, characterize situations and the precedence relation $\sqsubset$:

$$\neg s \sqsubset S_0 \tag{3.8}$$

$$s \sqsubset do(a, s') \equiv s \sqsubset s' \vee s = s' \tag{3.9}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2 \tag{3.10}$$

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s) \tag{3.11}$$

Axiom 3.8 states that S_0 has no preceding situations, thus, it is the initial situation; axiom 3.9 defines the precedence relation $\sqsubset$ i.e., ordering relation among situations; 3.10 is the unique names axiom for situations; and finally, axiom 3.11 states that for any property P, in order to show that $\forall s.P(s)$, it is sufficient to show that $P(S_0)$ holds, and inductively, for any situation s, if $P(s)$ holds then for any action a, $P(do(a, s))$ holds. It is a second-order induction axiom which has the effect of limiting the sort situation to the smallest set containing the initial situation S_0, and closed under the application of the function *do* to an action and a situation. Axioms 3.10 and 3.11 together imply that two situations will be the same if and only if they result from the same sequence of actions applied to the initial situation. Moreover, the domain of situations can be viewed as a tree whose root is S_0, and for each action a, $do(a, s)$ represents a child of s. As a result, for each situation s there is a unique finite sequence of actions $\vec{a}$ such that $s = do(\vec{a}, S_0)$.

Executable Situations and the Precedence Relation. When executability of situations is taken into consideration, it is possible to define $\leq$, a new precedence relation on situations that requires executability [131]. It is defined as:

$$s_1 \leq s_2 \doteq (s_1 = s_2 \vee (s_1 \sqsubset s_2 \wedge \forall a, s.(s_1 \sqsubset do(a, s) \sqsubseteq s_2 \supset Poss(a, s)))$$

where $s \sqsubseteq s'$ is an abbreviation for $s = s' \vee s \sqsubset s'$.

Relative Satisfiability Theorem. The property 3.5 leads to the *Relative Satisfiability Theorem* [132] which states that a BAT $\mathcal{D}$ is satisfiable if and only if $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$ is. This result ensures that provided the initial knowledge base together with the unique names axioms for actions are satisfiable, then unsatisfiability cannot be introduced by augmenting these with the foundational axioms for the situation calculus, together with action precondition and successor state axioms.

Additional constraints on BAT In this dissertation, we use a variation of the basic action theory described above which has more constraints imposed on it. It is defined as follows:

$$\mathcal{D} = \mathcal{D}_{poss} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{S_0} \cup \Sigma \cup \mathcal{D}_{ca} \cup \mathcal{D}_{coa} \tag{3.12}$$

Here, Σ, $\mathcal{D}_{poss}$, $\mathcal{D}_{ssa}$ and $\mathcal{D}_{S_0}$ are defined as above. In the following we describe $\mathcal{D}_{ca}$ and $\mathcal{D}_{coa}$ in more detail.

Unique Name Axioms for Actions and Domain Closure on Action Types $\mathcal{D}_{ca}$ The unique name axioms for actions is similar to $\mathcal{D}_{una}$. The domain closure axiom states that there are only the finitely many action types $A_1(\vec{x_1}), \ldots, A_n(\vec{x_n})$:

$$\forall a.[\exists \vec{x_1}.a = A_1(\vec{x_1}) \vee \ldots \vee \exists \vec{x_n}.a = A_n(\vec{x_n})] \tag{3.13}$$

This constraint eliminates the need to deal with existential quantification over action variables since we can replace it with quantification over objects for the finite action types.

Unique Name Axioms and Domain Closure for Object Constants $\mathcal{D}_{coa}$ Similar to Levesque and Lakemeyer [102], we assume the application domain is considered to be isomorphic to the set of standard object names $\mathcal{N} = \{\#0, \#1, \#2, \ldots\}$. One of the main consequences of this assumption is that quantification can be considered *substitutionally*; for instance, $\exists x.P(x)$ is true just in case $P(n)$ is true for some standard name n.

In [102] it is shown that one can an add infinite set of unique name axioms and an inference rule to get a complete axiomatization. Alternatively, one can axiomatize this constraint by providing axioms similar to second order arithmetic; in this alternative way we have a new function symbol *succ* and #0 and $\#1 \doteq succ(\#0)$, $\#2 \doteq succ(succ(\#0))$, etc., and we have the following second order axioms:

$$\begin{aligned}\forall x.succ(x) \neq \#0\\ \forall x, y.succ(x) = succ(y) \supset x = y\\ \forall P.[P(\#0) \wedge \forall x(P(x) \supset P(succ(x)))] \supset \forall x.P(x)\end{aligned} \tag{3.14}$$

We assume that $\mathcal{D}_{coa}$ is such an axiomatization. Note that the BAT may not use the function symbol *succ* explicitly only #0, #1, ... and quantification over objects.

3.1.3 Reasoning Tasks

Given an action theory containing facts representing a specific domain, there are various reasoning tasks that can be considered. The main one is called the *projection task*, which, given a sequence of actions and some initial situation, determines if a condition would hold if those actions were performed starting in that initial situation. More formally, given a basic action theory $\mathcal{D}$, a sequence of ground action terms $\vec{a}$, and a formula $\phi[s]$ that is uniform in s, the projection task determines whether or not

$$\mathcal{D} \models \phi(do(\vec{a}, S_0)) \tag{3.15}$$

Solutions to the projection include *regression* [132] and progression [132].

The *legality task* determines whether a sequence of actions *can* in fact be performed starting in some initial state, in other words, whether a sequence of actions leads to an executable situation. More formally, given a basic action theory $\mathcal{D}$, a sequence of ground actions terms $\vec{a}$, the legality task determines whether or not

$$\mathcal{D} \models Executable(do(\vec{a}, S_0)) \tag{3.16}$$

The *planning task* determines whether given a goal formula, it is possible to find a sequence of actions such that it follows from the basic action theory that the goal formula will hold in the situation that results from executing the actions in sequence starting in the initial state, and moreover, each action's preconditions are satisfied. More formally, given a basic action theory $\mathcal{D}$, and a situation-suppressed goal formula ϕ, the planning task is to find a ground action sequence $\vec{a}$ such that

$$\mathcal{D} \models Executable(do(\vec{a}, S_0)) \wedge \phi(do(\vec{a}, S_0)). \tag{3.17}$$

3.2 High-Level Programs

The situation calculus deals with specification and reasoning about (sequences of) primitive actions. *Complex actions*, that is, actions that have other actions as components including usual programming constructs such as conditionals, iteration, or recursive procedures are not considered. The most well known high-level programming language based on the situation calculus that allows representing and reasoning about complex actions (in addition to primitive actions) is Golog [103]. In Golog, the agent's knowledge of the domain dynamics (basic action theory), is specified *declaratively.* Then, a specification of the behavior of the agent in the domain (i.e., domain processes) is described *procedurally* by using complex actions. Such complex actions (i.e., programs) are formed from primitive actions or other complex actions using constructs like sequence, conditional, tests, iteration, non-deterministic branch, and non-deterministic choice of arguments. As a result of its logical foundations, Golog is able to accommodate *incomplete information*, either due to the fact that the initial state of the system is not completely specified (and hence this incomplete theory can result in several models), or because the non-deterministic constructs allow the program to evolve in any number of ways.

Several extensions of Golog exist, including Concurrent Golog (ConGolog) [33] which augments Golog with concurrency and interrupts, IndiGolog [34] which provides means for interleaving planning, sensing, and execution, and Decision-Theoretic Golog (DT-Golog) [19] that extends Golog to deal with quantified uncertainty (modeled probabilistically) in state specifications and action outcomes, and general reward functions. In the next section we look at the ConGolog agent programming language in more detail.

3.2.1 ConGolog

ConGolog [33] extends Golog with concurrent processes with possibly different priorities, high-level interrupts, and arbitrary exogenous actions. In this dissertation we concentrate on a fragment of ConGolog that includes the following constructs:

$$\delta ::= nil \mid \alpha \mid \varphi? \mid \delta_1;\delta_2 \mid \delta_1|\delta_2 \mid \pi x.\delta \mid \delta^* \mid \delta_1\|\delta_2 \mid \mathbf{atomic}(\delta) \mid \delta_1\&\ \delta_2 \mid \mathbf{set}(E) \mid \Sigma_s(\delta) \tag{3.18}$$

In the above, nil is a special program, called the *empty program*, that indicates the fact that nothing remains to be performed. An action term is represented by α, which possibly has parameters. A situation-suppressed formula, i.e., a formula with all situation arguments in fluents suppressed, is denoted by φ; also sometimes the situation argument is replaced by a placeholder *now* which represents the current situation. The formula obtained from φ by restoring the situation argument s into all fluents in φ is represented by $\varphi[s]$. E is a set of ground action sequences. δ_1 and δ_2 represent complex actions (i.e., programs). The test action $\varphi?$ determines if condition φ holds. The sequence of program δ_1 followed by program δ_2 is denoted by $\delta_1;\delta_2$. Program $\delta_1|\delta_2$ allows for the non-deterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* non-deterministic choice of a legal binding for variable x (observe that such a choice is, in general, unbounded). δ^* performs δ zero or more times. Program $\delta_1\|\delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2, while $\mathbf{atomic}(\delta)$ performs δ as an atomic unit, without allowing any interleaved actions [41]. The intersection/synchronous concurrent execution of programs δ_1 and δ_2 is denoted by $\delta_1\&\ \delta_2$ [35]. The $\mathbf{set}(E)$ construct is an infinitary non-deterministic branch; it takes an arbitrary set of sequences of actions E and turns it into a program [35]. Finally, the search operator $\Sigma_s(\delta)$ is used to specify that lookahead is performed over the (non-deterministic) program δ to ensure that non-deterministic choices are resolved in a way that ensures its successful completion [34].

In ConGolog [33], as concurrency is taken into account, a *single-step transition semantics* for programs is defined. This semantics is axiomatized through two predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$. $Trans(\delta, s, \delta', s')$ holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; in other words, *Trans* represents a transition relation between *configurations* (i.e., pairs formed by a program and a situation) (δ, s) and (δ', s'). $Final(\delta, s)$ holds if program δ may legally terminate in situation s.

Using the reflexive transitive closure of *Trans* denoted by $Trans^*$, it is possible to define the abbreviation $Do(\delta, s, s') \overset{\text{def}}{=} \exists\delta'. Trans^*(\delta, s, \delta', s') \land Final(\delta', s')$. The abbreviation $Do(\delta, s, s')$ denotes that the program δ, starting execution in s may legally terminate in situation s'. As complex actions may be non-deterministic, that is, they may have several different executions terminating in different situations, there may be several such s'.

The axioms for *Trans* and *Final* for the constructs in 3.18 (excluding **atomic**(), & and **set**) are shown below:

$$\begin{aligned}
&Trans(nil, s, \delta', s') \equiv \texttt{False}\\
&Trans(\alpha, s, \delta', s') \equiv s' = do(\alpha, s) \land Poss(\alpha, s) \land \delta' = \texttt{True}?\\
&Trans(\varphi?, s, \delta', s') \equiv \texttt{False}\\
&Trans(\delta_1; \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta'_1, s') \land \delta' = \delta'_1; \delta_2 \lor Final(\delta_1, s) \land Trans(\delta_2, s, \delta', s')\\
&Trans(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv \varphi[s] \land Trans(\delta_1, s, \delta', s') \lor \neg\varphi[s] \land Trans(\delta_2, s, \delta', s')\\
&Trans(\textbf{while } \varphi \textbf{ do } \delta, s, \delta', s') \equiv \varphi[s] \land Trans(\delta, s, \delta'', s') \land \delta' = \delta''; (\textbf{while } \varphi \textbf{ do } \delta)\\
&Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \lor Trans(\delta_2, s, \delta', s')\\
&Trans(\pi x.\delta, s, \delta', s') \equiv \exists x. Trans(\delta, s, \delta', s')\\
&Trans(\delta^*, s, \delta', s') \equiv Trans(\delta, s, \delta'', s') \land \delta' = \delta''; \delta^*\\
&Trans(\delta_1 \| \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta'_1, s') \land \delta' = \delta'_1 \| \delta_2 \lor Trans(\delta_2, s, \delta'_2, s') \land \delta' = \delta_1 \| \delta'_2\\
&Trans(\Sigma_s(\delta), s, \Sigma_s(\delta'), s') \equiv Trans(\delta, s, \delta', s') \land \exists s''. Do(\delta', s', s'')
\end{aligned}$$

$$\begin{aligned}
&Final(nil, s) \equiv \texttt{True}\\
&Final(\alpha, s) \equiv \texttt{False}\\
&Final(\varphi?, s) \equiv \varphi[s]\\
&Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \land Final(\delta_2, s)\\
&Final(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \equiv \varphi[s] \land Final(\delta_1, s) \lor \neg\varphi[s] \land Final(\delta_2, s)\\
&Final(\textbf{while } \varphi \textbf{ do } \delta, s) \equiv \varphi[s] \land Final(\delta, s') \lor \neg\varphi[s]\\
&Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \lor Final(\delta_2, s)\\
&Final(\pi x.\delta, s) \equiv \exists x. Final(\delta, s)\\
&Final(\delta^*, s) \equiv \texttt{True}\\
&Final(\delta_1 \| \delta_2, s) \equiv Final(\delta_1, s) \land Final(\delta_2, s)\\
&Final(\Sigma_s(\delta), s) \equiv Final(\delta, s)
\end{aligned}$$

The axioms for *Trans* and *Final* for the first 10 constructs above are as in [138]; these are the ones introduced in [33] except that following [29], the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Therefore, it is a *synchronous* version of the original test construct (interleaving is not allowed).

Details of the axioms above are provided in [33, 138, 34]. As an example, we look at the *Trans* and *Final* for the sequence construct: for $Trans(\delta_1; \delta_2, s, \delta', s')$ we have that $(\delta_1; \delta_2, s)$ can evolve to $(\delta'_1; \delta_2, s')$, provided that (δ_1, s) can evolve to (δ'_1, s'); alternatively, if (δ_1, s) is a final configuration and (δ_2, s) can evolve to (δ'_2, s'), it can also evolve to (δ'_2, s'). For $Final(\delta_1; \delta_2, s)$ we have that $(\delta_1; \delta_2, s)$ can be considered completed if both (δ_1, s) and (δ_2, s) are final.

Also, as in [138], it is required that in programs of the form $\pi x.\delta$, the variable x occurs in some non-variable action term in δ; cases where x occurs only in tests or as an action itself are not allowed. As a result, $\pi x.\delta$ acts as a construct for making non-deterministic choices of action parameters (possibly constrained by tests). Given the above semantics, it is possible to define the conditional and while loop constructs in terms of other constructs: **if** φ **then** δ_1 **else** $\delta_2 \doteq [\varphi?;\delta_1] \mid [\neg\varphi?;\delta_2]$ and **while** φ **do** $\delta \doteq [\varphi?;\delta]^*;\neg\varphi?$.

The definitions of *Trans* and *Final* for the **atomic**(δ) construct are provided in Appendix B. We discuss the axioms of *Trans* and *Final* for the constructs $\delta_1\&\ \delta_2$ and **set**(E) in sections 3.2.2 and 3.3 respectively.

Here and in the rest, we use $\mathcal{C}$ to denote the axioms defining the ConGolog programming language.

3.2.2 Situation Determined Programs

In *situation-determined* (SD) programs [35], non-determinism is restricted so that the remaining program is a *function* of the action performed. More formally, a ConGolog program δ is situation-determined in a situation s if for every sequence of actions, the remaining program is uniquely determined by the resulting situation, i.e.,

$$SituationDetermined(\delta,s) \doteq \forall s',\delta',\delta''.\,Trans^*(\delta,s,\delta',s') \wedge Trans^*(\delta,s,\delta'',s') \supset \delta' = \delta'' \qquad (3.19)$$

For instance, the ConGolog program $a;(b \mid c)$ (assuming the actions involved are always executable) is situation-determined in situation S_0. There is a unique remaining program $(b \mid c)$ in situation $do(a,S_0)$ (and in the same way, for the other reachable situations). However, the program $(a;b) \mid (a;c)$ is not situation-determined in situation S_0, since after performing action a, and given only the situation $do(a,S_0)$, it is impossible to determine what the remaining program is (it could be b or c). Any ConGolog program can be made situation-determined by adding decision actions to non-deterministic choices [38]. For example, the above program $(a;b)|(a;c)$ can be made situation determined by adding distinct decision actions $(option1;a;b)|(option2;a;c)$.

In situation-determined programs, a run of a program starting in a given situation can be considered as simply a sequence of actions, as the starting program, starting situation, and actions performed, functionally determine all the intermediate remaining programs that are gone through during the execution. Thus it is possible to view a program in a situation as specifying a *language* formed by all the sequences of actions that are runs of the program in the situation.[18] In this way, language theoretic notions such as union, intersection, and difference/complementation can be defined in terms of operations on the corresponding programs. The

[18]Note that in many cases not only the language will be infinite, but the alphabet on which the language is defined will be infinite, since it is formed by all action instances obtained by substituting values from the (possibly infinite) domain for parameters in the action types.

non-deterministic branch construct can be viewed as a union operator; and the intersection and difference of programs δ_1 and δ_2 can be defined as follows:[19]

$$
\begin{aligned}
Trans(\delta_1 \mathbin{\&} \delta_2, s, \delta', do(a,s)) \equiv{} & Trans(\delta_1, s, \delta'_1, do(a,s)) \wedge Trans(\delta_2, s, \delta'_2, do(a,s)) \wedge \delta' = \delta'_1 \mathbin{\&} \delta'_2 \\
Trans(\delta_1 - \delta_2, s, \delta', do(a,s)) \equiv{} & Trans(\delta_1, s, \delta'_1, do(a,s)) \wedge Trans(\delta_2, s, \delta'_2, do(a,s)) \wedge \delta' = \delta'_1 - \delta'_2 \vee \\
& Trans(\delta_1, s, \delta'_1, do(a,s)) \wedge \neg Trans(\delta_2, s, \delta'_2, do(a,s)) \wedge \delta' = \delta'_1
\end{aligned}
$$

$$
\begin{aligned}
& Final(\delta_1 \mathbin{\&} \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
& Final(\delta_1 - \delta_2, s) \equiv Final(\delta_1, s) \wedge \neg Final(\delta_2, s)
\end{aligned}
$$

In this context, it is possible to distinguish different types of runs for programs executing *offline* (where the agent does not acquire new knowledge during a run): the set $\mathcal{RR}_{\mathit{offl}}(\delta, s)$ of *partial runs* (0 or more steps) of a program δ in a situation s is defined as the sequences of actions that can be produced by executing δ from s:[20]

$$\mathcal{RR}_{\mathit{offl}}(\delta, s) = \{\vec{a} \mid \exists \delta'.\, Trans^*(\delta, s, \delta', do(\vec{a}, s))\} \tag{3.20}$$

where $\vec{a}$ is a sequence of actions; similarly, it is possible to define *complete runs*:

$$\mathcal{CR}_{\mathit{offl}}(\delta, s) = \{\vec{a} \mid \exists \delta'.\, Trans^*(\delta, s, \delta', do(\vec{a}, s)) \wedge Final(\delta', do(\vec{a}, s))\} \tag{3.21}$$

i.e., runs that end in a *Final* configuration, and *good runs*

$$\mathcal{GR}_{\mathit{offl}}(\delta, s) = \{\vec{a} \mid \exists \delta', \vec{b}.\, Trans^*(\delta, s, \delta', do(\vec{a}\vec{b}, s)) \wedge Final(\delta', do(\vec{a}\vec{b}, s))\} \tag{3.22}$$

i.e., partial runs that can be extended to a complete run.[21]

3.3 Agent Supervision

Agent supervision [35], proposed by De Giacomo, Lespérance and Muise (DLM), is a form of control / customization of the agent's behavior that ensures conformance to specifications while preserving the agent's autonomy. This framework is inspired by supervisory control of discrete event systems (SCDES) [166, 24], and is based on the situation calculus and situation-determined ConGolog. DLM's account of agent supervision is based on offline executions where an agent does not acquire new knowledge during a run.

[19]In sections 3.2.2 and 3.3 we change the notation slightly from [35] to promote clarity.

[20]The set notation is used for readability; it is possible to introduce $\mathcal{RR}_{\mathit{offl}}$ as a defined predicate [35].

[21]Note that quantification over sequence of actions (e.g., $\vec{b}$) is used for readability; it is possible to reformulate quantification over sequences of actions as quantification over situations in the situation calculus.

In this framework, the agent's possible behaviors are represented by a (non-deterministic) situation-determined ConGolog program δ_i relative to a BAT $\mathcal{D}$. The supervision specification is represented by another situation-determined ConGolog program δ_s.[22] In case it is possible to control all the actions of the agent, then it is straightforward to specify the result of supervision as the intersection /synchronous concurrent execution of the agent and the specification processes (δ_i & δ_s). Note that unless δ_i & δ_s is non-blocking (i.e., every partial run is a good run: $\mathcal{RR}_{\mathit{offl}}(\delta, s) = \mathcal{GR}_{\mathit{offl}}(\delta, s)$, or in other words, the execution can always be extended to a final state), it may become stuck in dead-end configurations; thus, it is assumed that the search construct Σ_s [45] is used in this case ($\Sigma_s(\delta_i \ \&\ \delta_s)$) which does lookahead to ensure that only good runs are considered.

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 notion is modeled by the special fluent $A_u(a, s)$ that means action a is *uncontrollable* in situation s.

DLM define the *controllability* of a supervision specification δ_s with respect to the agent program δ_i in situation s as:[23]

$$\begin{aligned} &\mathit{Controllable}(\delta_s, \delta_i, s) \doteq \\ &\forall \vec{a} a_u . \exists \vec{b} . \mathit{Do}(\delta_s, s, \mathit{do}([\vec{a}, \vec{b}], s)) \land A_u(a_u, \mathit{do}(\vec{a}, s)) \supset \\ &(\exists \vec{b} . \mathit{Do}(\delta_i, s, \mathit{do}([\vec{a}, a_u, \vec{b}], s)) \supset \exists \vec{b} . \mathit{Do}(\delta_s, s, \mathit{do}([\vec{a}, a_u, \vec{b}], s))), \end{aligned} \tag{3.23}$$

i.e., if we postfix an action sequence $\vec{a}$ that is good offline run for δ_s (i.e., such that $\exists \vec{b} . \mathit{Do}(\delta_s, s, \mathit{do}([\vec{a}, \vec{b}], s))$ holds) with an uncontrollable action a_u which is good for δ_i, then a_u must also be good for δ_s.

Then, DLM define the *offline maximally permissive supervisor* (offline MPS) $\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$ of the agent behavior δ_i which fulfills the supervision specification δ_s as:

$$\begin{aligned} &\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s) = \mathbf{set}(\bigcup_{E \in \mathcal{E}} E) \text{ where} \\ &\mathcal{E} = \{E \mid \forall \vec{a} \in E \supset \mathit{Do}(\delta_i \ \&\ \delta_s, s, \mathit{do}(\vec{a}, s)) \\ &\text{and } \mathbf{set}(E) \text{ is controllable wrt } \delta_i \text{ in } s\} \end{aligned} \tag{3.24}$$

i.e., the offline MPS is the union of all sets of action sequences that are complete offline runs of both δ_i and δ_s (i.e., such that $\mathit{Do}(\delta_i \ \&\ \delta_s, s, \mathit{do}(\vec{a}, s)))$ that are controllable for δ_i in situation s.

[22]In some cases, a declarative specification language would be preferable, e.g., linear temporal logic (LTL). Fritz and McIlraith [61] show how an extended version of LTL interpreted over a finite horizon can be compiled into ConGolog.

[23]In Section 6.1, we show that we can easily re-write this definition of controllability to avoid quantification over action sequences.

The above definition uses the $\mathbf{set}(E)$ construct introduced by DLM, which is a sort of infinitary non-deterministic branch; it takes an arbitrary set of sequences of actions E and turns it into a program. Its semantics are defined as follows:

$$\begin{aligned}
&\mathit{Trans}(\mathbf{set}(E), s, \delta', s') \equiv \exists a, \vec{a}.a\vec{a} \in E \land \mathit{Poss}(a, s) \land \\
&\qquad s' = do(a, s) \land \delta' = \mathbf{set}(\{\vec{a} \mid a\vec{a} \in E \land \mathit{Poss}(a, s)\}) \\
&\mathit{Final}(\mathbf{set}(E), s) \equiv \epsilon \in E
\end{aligned}$$

where ϵ is the empty sequence of actions. Therefore $\mathbf{set}(E)$ can be executed to produce any of the sequences of actions in E.[24]

DLM show that their notion of offline MPS, $\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$, has a number of important properties; it *always exists* and is *unique*, it is *controllable* with respect to the agent behavior δ_i in s, and it is the *largest* set of offline complete runs of δ_i that is controllable with respect to δ_i in s and satisfy the supervision specification δ_s in s, i.e., is *maximally permissive*:

Theorem 3.1 ([35]) *For the maximally permissive supervisor $\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$ the following properties hold:*

1. *$\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$ always exists and is unique;*
2. *$\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$ is controllable with respect to δ_i in s;*
3. *For every possible controllable supervision specification $\hat{\delta}_s$ for δ_i in s such that $\mathcal{CR}_{\mathit{offl}}(\delta_i \,\&\, \hat{\delta}_s, s) \subseteq \mathcal{CR}_{\mathit{offl}}(\delta_i \,\&\, \delta_s, s)$, we have that $\mathcal{CR}_{\mathit{offl}}(\delta_i \,\&\, \hat{\delta}_s, s) \subseteq \mathcal{CR}_{\mathit{offl}}(\delta_i \,\&\, \mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s), s)$.*

The infinitary program construct $\mathbf{set}(E)$ is mostly of theoretical interest, thus the above definition of $\mathit{mps}_{\mathit{offl}}(\delta_i, \delta_s, s)$ remains essentially mathematical. For practical purposes, a new construct $(\&_{A_u})$ for execution of programs under maximally permissive supervision is introduced by DLM. The construct $(\&_{A_u})$ is a special version of intersection that takes into account the fact that some actions are uncontrollable, and is characterized through *Trans* and *Final* as follows:

$$\begin{aligned}
&\mathit{Trans}(\delta_i \,\&_{A_u}\, \delta_s, s, \delta', do(a, s)) \equiv \\
&\quad \mathit{Trans}(\delta_i, s, \delta_i', do(a, s)) \land \mathit{Trans}(\delta_s, s, \delta_s', do(a, s)) \land \delta' = \delta_i' \,\&_{A_u}\, \delta_s' \text{ and} \\
&\quad \text{if } \neg A_u(a, s), \text{ then} \\
&\qquad \text{for all } \vec{a_u} \text{ such that } A_u(\vec{a_u}, do(a, s)) \\
&\qquad\quad \text{if } \exists \delta_i''.\mathit{Trans}^*(\Sigma_s(\delta_i), s, \delta_i'', do(a\vec{a_u}, s)), \\
&\qquad\quad \text{then } \exists \delta_s''.\mathit{Trans}^*(\Sigma_s(\delta_s), s, \delta_s'', do(a\vec{a_u}, s))
\end{aligned}$$

[24]Obviously there are certain sets that can be expressed directly in ConGolog, e.g., when E is finite. However in the general case, the object domain may be infinite, and $\mathbf{set}(E)$ may not be representable as a finitary ConGolog program.

where $A_u(\vec{a_u}, s)$, denotes that action sequence $\vec{a_u}$ is uncontrollable in situation s, and is inductively defined on the length of $\vec{a_u}$. Essentially, an action a by the agent is not allowed if it can be followed by some sequence of uncontrollable actions that violates the specification. *Final* for this new construct is defined as follows:

$$Final(\delta_i \,\&_{A_u}\, \delta_s, s) \equiv Final(\delta_i, s) \wedge Final(\delta_s, s)$$

i.e., $\delta_i \,\&_{A_u}\, \delta_s$ is final when both δ_i and δ_s are final. The $\&_{A_u}$ construct captures exactly the maximally permissive supervisor, and it is shown that:

Theorem 3.2 ([35]) $\mathcal{CR}_{\mathit{offl}}(\delta_i \,\&_{A_u}\, \delta_s, s) = \mathcal{CR}_{\mathit{offl}}(\delta_i \,\&\, mps_{\mathit{offl}}(\delta_i, \delta_s, s), s)$.

Note that while $mps_{\mathit{offl}}(\delta_i, \delta_s, s)$ is *always* non-blocking, $\delta_i \,\&_{A_u}\, \delta_s$ may not be. Hence, in general lookahead search must be used over the program $\delta_i \,\&_{A_u}\, \delta_s$ to find complete executions of $\&_{A_u}$. Therefore, $\Sigma_s(\delta_i \,\&_{A_u}\, \delta_s)$ is used.

In agent supervision, similarly to SCDES, the specification is separated from the system's behavior. This model supports system evolvability, especially in domains where user requirements and/or systems change frequently. In addition, it may be feasible to use such supervisors with existing (legacy) systems, provided minor changes (e.g., adding control points and observation capabilities) to such systems allows them to be controlled by the supervisor. Moreover, the supervising agent minimally restricts the behavior of the system agent, thus leaving it as much autonomy as possible to choose an action for execution among the set of allowed actions.

The agent supervision framework generalizes the SCDES in a number of ways. First, due to its first-order logic foundations, it enables a clear and formal characterization of the problem, copes with incomplete information, and can handle infinite states. Moreover, this approach allows techniques of reasoning about actions to be used, which enhances the process of decision making. Another noteworthy advantage of this framework is that it enables users to provide the system model and the specifications in a high-level expressive language. Finally, to provide greater flexibility in modeling systems, whether a primitive action is considered controllable is assumed to be situation-dependent.

4 Online Agent Supervision

The original DLM account of agent supervision assumes that the agent does not acquire new knowledge about her environment while executing. This means that all reasoning is done using the same knowledge base, and that only offline executions are considered.

In this chapter,[25] we study how we can apply the DLM framework in the case where the agent may acquire new knowledge while executing, for example through sensing. This means that the knowledge base that the agent uses in her reasoning needs to be updated during the execution. For instance, consider a travel planner agent that needs to book a seat on a certain flight. Only after querying the airline web service offering that flight will the agent know if there are seats available on the flight.

Technically, this requires switching from offline executions to *online executions* [39, 140], which, differently from offline executions, can only be defined meta-theoretically (unless one adds a knowledge operator/fluent [145]) since at every time point the knowledge base used by the agent to deliberate about the next action is different.

Based on online executions, we formalize the notion of *online maximally permissive supervisor* and show its existence and uniqueness, as in the simpler case of DLM. Moreover, we meta-theoretically define a program construct (i.e., supervision operator) for online supervised execution that given the agent and specification, executes them to obtain only runs allowed by the maximally permissive supervisor, showing its soundness and completeness. We also define a new lookahead search construct that ensures the agent can successfully complete the execution (i.e., ensures nonblockingness).

4.1 A Travel Planning Example

As a motivating example, we consider the task of planning a trip, which may involve booking a hotel and/or flight. Imagine we have a *very generic* travel agent whose behavior is defined by the ConGolog program $\delta_{travelPlanner}$ below, which we will later customize based on a client's requirements:

[25]The results of this chapter have already appeared in [14], [12], and [13].

```
$\delta_{travelPlanner}(cID, oC, dC, bD, eD) =$
  $(\delta_{qryHtl}(cID, dC, bD, eD) \mid \delta_{qryAir}(cID, oC, dC, bD, eD) \mid$
   $\delta_{pickHtl}(cID, dC, bD, eD) \mid \delta_{pickAir}(cID, oC, dC, bD, eD))^*;$
  if $\exists htlID\ SelHtl(cID, htlID) \vee \exists airID\ SelAir(cID, airID)$
   then [
    $displaySelectedProposals(cID);$
    $\pi htlID, airID.[clntResponse(cID, htlID, airID);$
    if $htlID \neq NULL$ then $bookHtl(cID, htlID);$
    if $airID \neq NULL$ then $bookAir(cID, airID)]$
  ] else
    $reportFailure(cID)$
  endIf
```

The argument cID stands for the client session ID, oC for the origin city, dC for destination city, and finally bD and eD represent the begin and end dates of the trip respectively. These are assumed to be known at the outset of the program. The program begins by querying 0 or more hotel and/or airline web services in any order, to check if they can fulfill the given request. The querying is done by two subprograms δ_{qryHtl} and δ_{qryAir}, with the former defined as follows:

```
$\delta_{qryHtl}(cID, dC, bD, eD) =$
  $(\pi htlID.[\neg QrdHtlWS(cID, htlID, dC, bD, eD)?;$
  $qryHtlWS(cID, htlID, dC, bD, eD);$
  $\pi hC, stat.replyHtlWS(cID, htlID, hC, stat)]$
```

This program can query any hotel web service ($htlID$), as long as it has not been queried before. The web service's reply to the query is represented by the exogenous action $replyHtlWS$, where the parameters hC and $stat$ return the cost and status of the web service (OK if it is able to fulfill the request and $NotAvail$ otherwise) respectively. The fluent $RpldHtl(cID, htlID, dC, bD, eD, hC, stat, s)$ stores the reply from the hotel web service after it is received. The procedure δ_{qryAir} is defined similarly and the fluent $RpldAir$ stores the reply from the airline web service.

As it is querying hotel and/or airline web services, $\delta_{travelPlanner}$ also executes the subprograms $\delta_{pickHtl}$ and $\delta_{pickAir}$ to select 0 or more hotel and/or airline web service IDs, among those that replied positively, in any order. The procedure $\delta_{pickHtl}$ is defined as follows:

$$
\begin{aligned}
&\delta_{pickHtl}(cID, oC, dC, bD, eD) = \\
&\quad ((\pi htlD, hC.[RpldHtl(cID, htlID, dC, bD, eD, hC, OK)?; \\
&\quad selectHtl(cID, htlID)])
\end{aligned}
$$

$\delta_{pickAir}$ is defined similarly. The fluents $SelHtl(cID, htlID, s)$ and $SelAir(cID, airID, s)$ become true for the selected hotels and airlines after actions $selectHtl(cID, htlID)$ and $selectAir(cID, airID)$ respectively.

If some hotels and/or airlines are selected, the program then presents them to user by executing the action $displaySelectedProposals(cID)$ after which the fluents $ProposedHtl(cID, htlID, s)$ and $ProposedAir(cID, airID, s)$ become true for those hotels and airlines. Subsequently, the client's response, represented by the exogenous action $clntResponse(cID, htlID, airID)$ is obtained, where $htlID$ and $airID$ may contain the chosen hotel and airline or $NULL$. Finally, the program will book the hotel and/or airline chosen by the client, if any. The fluents $BookedHtl(cID, htlID, s)$ and/or $BookedAir(cID, htlID, s)$ become true for the hotel and/or airline booked for the client. The program may also simply report failure without selecting any hotels and airlines.

In addition to a client's basic requirements such as begin and end date of the trip, the client may have further constraints and preferences. For instance, he may not want to fly with a certain airline, or he may want the travel planner to propose at least two hotels, if possible. He may also have budget constraints. Such constraints and preferences can be represented by another ConGolog program. Then, *supervision* can be utilized to personalize the travel planner program based on the given specification. For example, suppose that the client wants a hotel but not $HtlX$. We can represent this constraint by the following supervision specification:

$$
\begin{aligned}
&\delta_{client1}(cID, oC, dC, bD, eD) = \\
&\quad [(\pi a.a)^*; \\
&\quad \exists htlID.\ ProposedHtl(cID, htlID) \vee \\
&\quad \forall htlID.IsHtl(htlID, dc) \wedge htlID \neq HtlX \supset \\
&\quad RpldHtl(cID, htlID, dC, bD, eD, hC, NotAvail)?; \\
&\quad (\pi a.a)^*]\ \& \\
&\quad \pi a.(a; \neg BookedHtl(cID, HtlX)?)^*
\end{aligned}
$$

This specification is satisfied if eventually (i.e., after some sequence of actions) a hotel has been proposed or all hotels other than $HtlX$ replied that no room is available, and $HtlX$ is never booked. In the above, $IsHtl$ is a relation that holds for all hotels in the destination city.

What executions can we get if we perform supervision on the generic travel planner agent with this specification? Suppose for simplicity that there are only three hotel web services ($HtlX$, $HtlY$, and $HtlZ$) and no airline web services. The supervision does not enforce any specific order in which these hotel web services are queried. Assuming the program queries $HtlY$ first and it is not able to fulfill the request, then $HtlZ$ must be queried. The program can only report failure if neither of $HtlY$ and $HtlZ$ have available rooms. In case $HtlY$ is able to fulfill the request, then $HtlZ$ may be queried. In either case, $HtlX$ may be queried as well. Once $HtlY$ and/or $HtlZ$ indicated availability, the travel planner must then select and present at least one of them to the client. After obtaining the client's response, the program must book the client's chosen hotel, if any. Supervision must ensure that $HtlX$ is not selected and proposed to the client. If it is, it might be chosen by the client, meaning that the agent must book it, which clearly violates the supervision specification. At the same time, supervision must leave as much freedom for the program as possible, so for example, the program can still query $HtlX$ and need not query $HtlZ$ in case $HtlY$ has a room available. What supervision produces will become clearer after we have given the formal definitions in Section 4.3. We will return to our example there.

4.2 Agents Executing Online

In our account of agent supervision, we want to accommodate agents that can acquire new knowledge about their environment during execution, for example by sensing, and where their knowledge base is updated with this new knowledge. Thus we consider an agent's *online executions*, where, as she executes the program, at each time point, she makes decisions on what to do next based on what her current knowledge is.

4.2.1 Sensing and Exogenous Actions

A crucial aspect of online executions is that the agent can take advantage of sensing. Similarly to the approach of [99] (see Section 2.3), we model sensing as an ordinary action which queries a sensor, followed by the reporting of a sensor result, in the form of an exogenous action.

Specifically, to sense whether fluent P holds within a program, we use a macro:

$$SenseP \doteq QryIfP; (repValP(1) \mid repValP(0)),$$

where $QryIfP$ is an ordinary action that is always executable and is used to query (i.e., sense) if P holds and $repValP(x)$ is an exogenous action with no effect that informs the agent if P holds through its precondition axiom, which is of the form:

$$Poss(repValP(x), s) \equiv P(s) \land x = 1 \lor \neg P(s) \land x = 0.$$

Thus, we can understand that $SenseP$ reports value 1 through the execution of $repValP(1)$ if P holds, and 0 through the execution of $repValP(0)$ otherwise.

For example, consider the following agent program:

$$\delta^i = SenseP; [P?; A] \mid [\neg P?; B]$$

and assume the agent does not know if P holds initially. So initially we have $\mathcal{D} \cup \mathcal{C} \models Trans(\delta^i, S_0, \delta', S_1)$ where $S_1 = do(QryIfP, S_0)$ and $\delta' = nil; (repValP(1) \mid repValP(0))); [P?; A] \mid [\neg P?; B]$. At S_1, the agent knows either of the exogenous actions $repValP(0)$ or $repValP(1)$ could occur, but does not know which. After the occurrence of one of these actions, the agent learns whether P holds. For example, if $repValP(1)$ occurs, the agent's knowledge base is now updated to $\mathcal{D} \cup \mathcal{C} \cup \{Poss(repValP(1), S_1)\}$ (where $\mathcal{C}$ denotes the axioms defining the ConGolog programming language). With this updated knowledge, she knows which action to do next:

$$\begin{aligned}&\mathcal{D} \cup \mathcal{C} \cup Poss(repValP(1), S_1) \models\\ &\quad Trans(nil; [P?; A] \mid [\neg P?; B], do(repValP(1), S_1), nil, do([repValP(1), A], S_1)).\end{aligned}$$

Notice that with this way of doing sensing, we essentially store the sensing results in the situation (which includes all actions executed so far including the exogenous actions used for sensing). In particular the current KB after having performed the sequence of actions $\vec{a}$ is:

$$\mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\}.$$

Note that this approach also handles the agent's acquiring knowledge from an arbitrary exogenous action.

4.2.2 Online Executions

Agent online configurations and transitions. We denote an *agent* by σ, which stands for a pair $\langle \mathcal{D}, \delta^i \rangle$, where δ^i is the initial program of the agent expressed in ConGolog and $\mathcal{D}$ is a BAT that represents the agent's initial knowledge (which may be incomplete). We assume that we have a finite set of primitive action types $\mathcal{A}$, which is the disjoint union of a set of ordinary primitive action types $\mathcal{A}^o$ and exogenous primitive action types $\mathcal{A}^e$.

An *agent configuration* is modeled as a pair $\langle \delta, \vec{a} \rangle$, where δ is the remaining program and $\vec{a}$ is the current history, i.e, the sequence of actions performed so far starting from S_0. The initial configuration c^i is $\langle \delta^i, \epsilon \rangle$, where ϵ is the empty sequence of actions.

The *online transition relation* between agent configurations is (a meta-theoretic) binary relation between configurations defined as follows:

$$\begin{array}{l}\langle \delta, \vec{a}\rangle \rightarrow_{A(\vec{n})} \langle \delta', \vec{a}A(\vec{n})\rangle \\ \qquad \text{if and only if} \\ \text{either } A \in \mathcal{A}^o,\ \vec{n} \in \mathcal{N}^k \text{ and} \\ \quad \mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Trans(\delta, do(\vec{a}, S_0), \delta', do(A(\vec{n}), do(\vec{a}, S_0))) \\ \text{or } A \in \mathcal{A}^e,\ \vec{n} \in \mathcal{N}^k \text{ and} \\ \quad \mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)), Trans(\delta, do(\vec{a}, S_0), \delta', do(A(\vec{n}), do(\vec{a}, S_0)))\} \text{ is satisfiable.}\end{array}$$

Here, $\langle \delta, \vec{a}\rangle \rightarrow_{A(\vec{n})} \langle \delta', \vec{a}A(\vec{n})\rangle$ means that configuration $\langle \delta, \vec{a}\rangle$ can make a single-step online transition to configuration $\langle \delta', \vec{a}A(\vec{n})\rangle$ by performing action $A(\vec{n})$. If $A(\vec{n})$ is an ordinary action, the agent must know that the action is executable and know what the remaining program is afterwards. If $A(\vec{n})$ is an exogenous action, the agent need only think that the action may be possible with δ' being the remaining program, i.e., it must be consistent with what she knows that the action is executable and δ' is the remaining program. As part of the transition, the theory is (implicitly) updated in that the new exogenous action $A(\vec{n})$ is added to the action sequence, and $Executable(do([\vec{a}, A(\vec{n})], S_0))$ will be added to the theory when it is queried in later transitions, thus incorporating the fact that $Poss(A(\vec{n}), do(\vec{a}, S_0))$ is now known to hold.

The (meta-theoretic) relation $c \rightarrow^*_{\vec{a}} c'$ is the reflexive-transitive closure of $c \rightarrow_{A(\vec{n})} c'$ and denotes that online configuration c' can be reached from the online configuration c by performing a sequence of online transitions involving the sequence of actions $\vec{a}$.

We also define a (meta-theoretic) predicate $c^{\checkmark}$ meaning that the online configuration c is known to be final:

$$\begin{array}{l}\langle \delta, \vec{a}\rangle^{\checkmark} \text{ if and only if} \\ \mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Final(\delta, do(\vec{a}, S_0)).\end{array}$$

Online situation determined agents. Here, we are interested in programs that are situation-determined (SD), i.e., given a program, a situation and an action, we want the remaining program to be determined. However this is not sufficient when considering online executions. We want to ensure that the agent always knows what the remaining program is after any sequence of actions. We say that an agent is *online situation-determined* (online SD) if for any sequence of actions that the agent can perform online, the resulting agent configuration is unique. Formally, an agent $\sigma = \langle \mathcal{D}, \delta^i\rangle$ with initial configuration $c^i = \langle \delta^i, \epsilon\rangle$ is *online SD* if and only if for all sequences of actions $\vec{a}$, if $c^i \rightarrow^*_{\vec{a}} c'$ and $c^i \rightarrow^*_{\vec{a}} c''$ then $c' = c''$.

We say that an agent $\sigma = \langle \mathcal{D}, \delta^i\rangle$ *always knows the remaining program after an exogenous action* if and only if for all action sequences $\vec{a}, A \in \mathcal{A}^e,\ \vec{n} \in \mathcal{N}^k$

if $\quad \mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)), Trans(\delta, do(\vec{a}, S_0), \delta', do([\vec{a}, A(\vec{n})], S_0))\}$ is satisfiable,

then there exists a program δ' such that

$\mathcal{D} \cup \mathcal{C} \cup \{Executable(do([\vec{a}, A(\vec{n})], S_0))\} \models Trans(\delta, do(\vec{a}, S_0), \delta', do([\vec{a}, A(\vec{n})], S_0))$.

Essentially, it states that whenever the agent considers it possible that an exogenous action may occur, then she knows what the remaining program is afterwards if it does occur.

We can show that (for proofs of our results in this chapter, see Appendix A.1):

Theorem 4.1 *For any agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$, if δ^i is known to be SD in $\mathcal{D}$, i.e., $\mathcal{D} \cup \mathcal{C} \models SituationDetermined(\delta^i, S_0)$, and if σ always knows the remaining program after an exogenous action, then σ is online SD.*

Being online SD is an important property. It means that for any sequence of actions that the agent can perform in an online execution, there is a unique resulting agent configuration, i.e., agent belief state and remaining program. From now on, we assume that the agent is online SD.

Online Runs. For an agent σ that is online SD, online executions can be succinctly represented by *runs* formed by the corresponding sequence of actions. The set $\mathcal{RR}(\sigma)$ of (partial) *runs* of an online SD agent σ with starting configuration c^i is the sequences of actions that can be produced by executing c^i from S_0: $\mathcal{RR}(\sigma) = \{\vec{a} \mid \exists c. c^i \rightarrow^*_{\vec{a}} c\}$. A run is *complete* if it reaches a final configuration. Formally we define the set $\mathcal{CR}(\sigma)$ of complete runs as: $\mathcal{CR}(\sigma) = \{\vec{a} \mid \exists c. c^i \rightarrow^*_{\vec{a}} c \wedge c^{\checkmark}\}$. Finally we say that a run is *good* if it can be extended to a complete run. Formally we define the set $\mathcal{GR}(\sigma)$ of good runs as: $\mathcal{GR}(\sigma) = \{\vec{a} \mid \exists c, c', \vec{a'}. c^i \rightarrow^*_{\vec{a}} c \wedge c \rightarrow^*_{\vec{a'}} c' \wedge c'^{\checkmark}\}$.

4.3 Online Agent Supervision

4.3.1 Motivation

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. DLM's account of agent supervision is based on offline executions and does not accommodate agents that acquire new knowledge during a run.

To see why the notion of offline MPS is inadequate in the context of online execution, let's look at the following example:

Example 4.1 Suppose that we have an agent that does not know whether P holds initially, i.e., $\mathcal{D} \not\models P(S_0)$ and $\mathcal{D} \not\models \neg P(S_0)$. Suppose that the agent's initial program is:

$$\delta_4^i = [P?; ((A; (C \mid U)) \mid (B; D))] \ \mid [\neg P?; ((A; D) \mid (B; (C \mid U)))]$$

where all actions are ordinary, always executable, and controllable except for U, which is always uncontrollable. Suppose that the supervision specification is:

$$\delta_4^s = (\pi a.a \neq U?; a)^*$$

i.e., any action except U can be performed. It is easy to show that the offline MPS obtained using DLM's definition is different depending on whether P holds or not:

$$\mathcal{D} \cup \mathcal{C} \models (P(S_0) \supset mps_{\mathit{offl}}(\delta_4^i, \delta_4^s, S_0) = \mathbf{set}(\{[B, D]\})) \wedge \\ (\neg P(S_0) \supset mps_{\mathit{offl}}(\delta_4^i, \delta_4^s, S_0) = \mathbf{set}(\{[A, D]\}))$$

where mps_{offl} is defined as in Section 3.3. For models of the theory where P holds, the offline MPS is $\mathbf{set}(\{B, D\})$, as the set of complete offline runs of δ_4^s in S_0 is $\{[B, D], [A, C]\}$ and $\mathbf{set}(\{[A, C]\})$ is not controllable with respect to δ_4^i in S_0. For models where P does not hold, the offline MPS is $\mathbf{set}(\{A, D\})$, since the set of complete offline runs of δ_4^s in S_0 is $\{[A, D], [B, C]\}$ and $\mathbf{set}(\{[B, C]\})$ is not controllable with respect to δ_4^i in S_0. Since it is not known if P holds, it seems that a correct supervisor should neither allow A nor B. $\square$

As the above example illustrates, we have an offline MPS for each model of the theory. Instead, we want a single online MPS that works for all models and includes sensing information when acquired. The difference between offline MPS and online MPS is analogous to the difference between classical plans and conditional plans that include sensing in the planning literature [120, 121].

4.3.2 Online Maximally Permissive Supervisor

In our account of supervision, we want to deal with agents that may acquire knowledge through sensing and exogenous actions as they operate and make decisions based on what they know, and we model these as online SD agents. Let's see how we can formalize supervision for such agents. Assume that we have an online SD agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$ whose behavior we want to supervise. Let's also suppose that we have a *supervision specification* δ^s of what behaviors we want to allow in the supervised system, where δ^s is a SD ConGolog program relative to the BAT $\mathcal{D}$ of the agent. In fact, we assume that the system $\langle \mathcal{D}, \delta^s \rangle$ is also online SD. We say that a specification δ^s is *online controllable* with respect to online SD agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$ iff:

$$\begin{array}{l}\forall \vec{a}a_u.\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle) \text{ and} \\ \mathcal{D} \cup \{Executable(do(\vec{a}, S_0))\} \not\models \neg A_u(a_u, do(\vec{a}, S_0)) \text{ implies} \\ \quad \text{if } \vec{a}a_u \in \mathcal{GR}(\sigma) \text{ then } \vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle).\end{array}$$

i.e., if we postfix a good online run $\vec{a}$ for $\langle \mathcal{D}, \delta^s \rangle$ with an action a_u that is not known to be controllable which is good for σ (and so $\vec{a}$ must be good for σ as well), then a_u must also be good for $\langle \mathcal{D}, \delta^s \rangle$. (Note that $\vec{a}a_u \in \mathcal{GR}(\sigma)$ and $\vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$ together imply that $\vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \& \ \delta^s \rangle)$.) This definition is quite similar to DLM's. But it differs in that it applies to online runs as opposed to offline runs. Moreover it treats actions that are not known to be controllable as uncontrollable, thus ensuring that δ^s is controllable in all possible models/worlds compatible with what the agent knows. Note that like DLM, we focus on good runs of the program, assuming that the agent will not perform actions that don't lead to a final configuration of δ^i. The supervisor only ensures that given this assumption, the program always conforms to the specification.

We define the *online maximally permissive supervisor* $mps_{onl}(\delta^s, \sigma)$ of the online SD agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$ which fulfills the supervision specification δ^s:

$$\begin{array}{l} mps_{onl}(\delta^s, \sigma) = \mathbf{set}(\bigcup_{E \in \mathcal{E}} E) \text{ where} \\ \mathcal{E} = \{E \mid E \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ \delta^s \rangle) \text{ and } \mathbf{set}(E) \text{ is online controllable with respect to } \sigma\}\end{array}$$

i.e., the online MPS is the union of all sets of action sequences that are complete online runs of both δ^i and δ^s that are online controllable for the agent σ. Again, our definition is similar to DLM's, but applies to online runs, and relies on online (as opposed to offline) controllability. We can show that:

Theorem 4.2 *For the online maximally permissive supervisor $mps_{onl}(\delta^s, \sigma)$ of the online SD agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$ which fulfills the supervision specification δ^s, where $\langle \mathcal{D}, \delta^s \rangle$ is also online SD, the following properties hold:*

1. *$mps_{onl}(\delta^s, \sigma)$ always exists and is unique;*
2. *$\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle$ is online SD;*
3. *$mps_{onl}(\delta^s, \sigma)$ is online controllable with respect to σ;*
4. *for every possible online controllable supervision specification $\hat{\delta^s}$ for σ such that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \& \hat{\delta^s} \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \& \delta^s \rangle)$, we have that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \& \hat{\delta^s} \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$, i.e., mps_{onl} is maximally permissive;*
5. *$\mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle) = \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$, i.e., $mps_{onl}(\delta^s, \sigma)$ is non-blocking.*

Example 4.2 If we return to the agent of Example 4.1, who does not know whether P holds initially, it is easy to show that our definition of online MPS yields the correct result, i.e. $mps_{onl}(\delta_4^s, \langle \mathcal{D}, \delta_4^i \rangle) = \mathbf{set}(\{\epsilon\})$. $\square$

Example 4.3 Supervision can also depend on the information that the agent acquires as it executes. Again, suppose that we have an agent that does not know whether P holds initially. Suppose also that the agent's initial program is $\delta_5^i = Sense_P; \delta_4^i$. We can show that:

$$\begin{aligned}\mathcal{D} \cup \mathcal{C} \models (P(S_0) \supset mps_{\mathit{offl}}(\delta_5^i, \delta_4^s, S_0) = \\ \mathbf{set}(\{[QryIfP, repValP(1), B, D]\})) \wedge \\ (\neg P(S_0) \supset mps_{\mathit{offl}}(\delta_5^i, \delta_4^s, S_0) = \\ \mathbf{set}(\{[QryIfP, repValP(0), A, D]\}))\end{aligned}$$

Again, we have different offline MPSs depending on whether P holds. But since the exogenous report makes the truth value of P known after the first action, we get one online MPS for this agent as follows:

$$mps_{onl}(\delta_4^s, \langle \mathcal{D}, \delta_5^i \rangle) = \mathbf{set}(\{[QryIfP, repValP(1), B, D], [QryIfP, repValP(0), A, D]\})$$

Because the agent queries if P holds, the supervisor has enough information to decide the maximal set of runs from then on in each case. So if the reported value of P is true, then the online supervisor should eliminate the complete run $[A, C]$ as it is not controllable, and if P does not hold, the run $[B, C]$ should be eliminated for the same reason. $\square$

As well, an action's controllability or whether it satisfies the specification may depend on a condition whose truth only becomes known during the execution. Such cases cannot be handled by DLM's original offline account but our online supervision account does handle them correctly.

4.3.3 Online Supervision Operator

We can also introduce a meta-theoretic version of a synchronous concurrency operator $\delta^i \&_{A_u}^{onl} \delta^s$ that captures the *maximally permissive execution of an agent* $\langle \mathcal{D}, \delta^i \rangle$ *under online supervision for specification* δ^s. Without loss of generality, we assume that both δ^i and δ^s start with a common controllable action (if not, it is trivial to add a dummy action in front of both so as to fulfill the requirement). We define $\delta^i \&_{A_u}^{onl} \delta^s$ by extending the online transition relation as follows:

$$\begin{gathered}
\langle \delta^i \&_{A_u}^{onl} \delta^s, \vec{a} \rangle \rightarrow_a \langle \delta^{i'} \&_{A_u}^{onl} \delta^{s'}, \vec{a}a \rangle \\
\text{if and only if} \\
\langle \delta^i, \vec{a} \rangle \rightarrow_a \langle \delta^{i'}, \vec{a}a \rangle \text{ and } \langle \delta^s, \vec{a} \rangle \rightarrow_a \langle \delta^{s'}, \vec{a}a \rangle \text{ and} \\
\text{if } \mathcal{D} \cup \{Executable(do(\vec{a}, S_0))\} \models \neg A_u(a, do(\vec{a}, S_0)) \\
\text{then for all } \vec{a_u} \text{ s.t. } \mathcal{D} \cup \{Executable(do(\vec{a}a\vec{a}_u, S_0)), \\
\qquad A_u(\vec{a}_u, do(\vec{a}a, S_0))\} \text{ is satisfiable,} \\
\text{if } \vec{a}a\vec{a_u} \ \in \mathcal{GR}(\langle \mathcal{D}, [\vec{a}; \delta^i] \rangle), \text{then } \vec{a}a\vec{a_u} \ \in \mathcal{GR}(\langle \mathcal{D}, [\vec{a}; \delta^s] \rangle).
\end{gathered}$$

where $A_u(\vec{a_u}, s)$, means that action sequence $\vec{a_u}$ is uncontrollable in situation s, and is inductively defined on the length of $\vec{a_u}$ as the smallest predicate such that: *(i)* $A_u(\epsilon, s) \equiv \texttt{true}$; *(ii)* $A_u(a_u\vec{a_u}, s) \equiv A_u(a_u, s) \wedge A_u(\vec{a_u}, do(a_u, s))$. Thus, the online maximally permissive supervised execution of δ^i for the specification δ^s is allowed to perform action a in situation $do(\vec{a}, S_0)$ if a is allowed by both δ^i and δ^s and moreover, if a is known to be controllable, then for every sequence of actions $\vec{a_u}$ not known to be controllable, if $\vec{a_u}$ may be performed by δ^i right after a on one of its complete runs, then it must also be allowed by δ^s (on one of its complete runs). Essentially, a controllable action a by the agent must be forbidden if it can be followed by some sequence of actions not known to be controllable that violates the specification.

The final configurations are extended as follows:

$$(\langle \delta^i \&_{A_u}^{onl} \delta^s, \vec{a} \rangle)^{\checkmark} \text{ if and only if } (\langle \delta^i, \vec{a} \rangle)^{\checkmark} \text{ and } (\langle \delta^s, \vec{a} \rangle)^{\checkmark}$$

We can show that firstly, if both the agent and supervision specification programs are online SD, then so is the program obtained using the online supervision operator, and moreover, this program is controllable with respect to to the agent program:

Theorem 4.3

1. *If* $\langle \mathcal{D}, \delta^s \rangle$ *and* $\langle \mathcal{D}, \delta^i \rangle$ *are online SD, then so is* $\langle \mathcal{D}, \delta^i \&_{A_u}^{onl} \delta^s \rangle$.
2. $\delta^i \ \&_{A_u}^{onl} \ \delta^s$ *is online controllable with respect to* $\langle \mathcal{D}, \delta^i \rangle$.

Moreover, the complete runs of the program obtained using the online supervision operator are exactly the same the complete runs generated under synchronous concurrency of the agent and $mps_{onl}(\delta^s, \sigma)$:

Theorem 4.4

$$\mathcal{CR}(\langle \mathcal{D}, \delta^i \ \&_{A_u}^{onl} \ \delta^s \rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma) \rangle).$$

While $\delta^i \ \&_{A_u}^{onl} \ \delta^s$ and $mps_{onl}(\delta^s, \sigma)$ have the same complete runs, it is not the case that they have the same set of partial runs. In fact in general, $\mathcal{RR}(\langle \mathcal{D}, \delta^i \ \&_{A_u}^{onl} \ \delta^s \rangle) \neq \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \&_{A_u}^{onl} \ \delta^s \rangle)$, i.e., the program obtained using the online supervision operator is not necessarily non-blocking. This is in contrast with $mps_{onl}(\delta^s, \sigma)$, which is guaranteed to be non-blocking (Theorem 4.2).

Example 4.4 Suppose we have the agent program:

$$\delta_6^i = (A \mid [B; C; (U1 \mid U2; D)])$$

where all actions except $U1$ and $U2$ are ordinary and controllable. Moreover, assume the supervision specification is:

$$\delta_6^s = (\pi a.a \neq D?; a)^*$$

i.e., any action except D can be performed. The online MPS for this agent is simply $\mathbf{set}(\{A\})$, since $\mathcal{CR}(\langle \mathcal{D}, \delta_6^s \rangle) = \{A, [B, C, U1]\}$ and $\mathbf{set}(\{[B, C, U1]\})$ is not controllable with respect to δ_6^i. However, under online supervised execution, the agent may execute the action B. We have $\langle \delta_6^i \&_{A_u}^{onl} \delta_6^s, \epsilon \rangle \rightarrow_B \langle \delta_6' \&_{A_u}^{onl} \delta_6^s, B \rangle$ where δ_6' is what remains from δ_6^i after executing B. The resulting program is not final in $do(B, S_0)$, yet there is no transition from this state, as the action C could be followed by the uncontrollable action $U2$ and it is not possible to ensure successful completion of the program, as the action D is not allowed. Thus, one must do lookahead search over online executions of $\delta_6^i \;\&_{A_u}^{onl}\; \delta_6^s$ to obtain good/complete runs. We propose such a search/lookahead construct next.

□

4.3.4 Search Over a Controllable Program

When we have a specification/program δ^s that is controllable with respect to an agent $\langle \mathcal{D}, \delta^i \rangle$ (like for instance, $\delta^i \;\&_{A_u}^{onl}\; \delta^s$), for any choice of uncontrollable action that is on a good run of δ^i, it is always possible to find a way to continue executing δ^s until the program successfully completes. We can define a search construct[26] that is applicable to these cases. It makes an arbitrary choice of action that is on a good run of δ^i when the action is not known to be controllable, while still only performing actions that are on a good run of δ^s otherwise. We call this construct *weak online search* $\Sigma_{onl}^w(\delta^s, \delta^i)$ and define it (meta-theoretically) as follows:[27]

[26] In IndiGolog a simple type of search is provided that only allows a transition if the remaining program can be executed to reach a final state in all the offline executions [39]. However, this search does not deal with sensing and online executions.

[27]Since δ^i can include exogenous actions, in general, executions of the program could actually perform exogenous actions that are not on a good run of δ^i. However, here we are interested in the case where the exogenous actions are mainly sensor reports and external requests (rather than the actions of an adversary). Handling adversarial non-determinism in δ^i is left for future work.

$$\begin{gathered}\langle \Sigma^w_{onl}(\delta^s, \delta^i), \vec{a}\rangle \rightarrow_a \langle \Sigma^w_{onl}(\delta^{s'}, \delta^{i'}), \vec{a}a\rangle \\ \text{if and only if} \\ \langle \delta^s, \vec{a}\rangle \rightarrow_a \langle \delta^{s'}, \vec{a}a\rangle \text{ and } \langle \delta^i, \vec{a}\rangle \rightarrow_a \langle \delta^{i'}, \vec{a}a\rangle \text{ and} \\ \text{if } \mathcal{D} \cup \{Executable(\vec{a}, S_0)\} \models \neg A_u(a, do(\vec{a}, S_0)) \\ \text{then } \vec{a}a \in \mathcal{GR}(\langle \mathcal{D}, [\vec{a}a; \delta^{s'}]\rangle) \\ \text{else } \vec{a}a \in \mathcal{GR}(\langle \mathcal{D}, [\vec{a}a; \delta^{i'}]\rangle)\end{gathered}$$

The final configurations are extended as follows:

$$(\langle \Sigma^w_{onl}(\delta^s, \delta^i), \vec{a}\rangle)^{\checkmark} \text{ iff } (\langle \delta^s, \vec{a}\rangle)^{\checkmark} \text{ and } (\langle \delta^i, \vec{a}\rangle)^{\checkmark}$$

It is easy to show that:

Theorem 4.5 *If $\langle \mathcal{D}, \delta^s\rangle$ and $\langle \mathcal{D}, \delta^i\rangle$ are online SD, than so is $\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle$.*

Now, we can show that the weak online search construct has many nice properties when the program is controllable:

Theorem 4.6 *Suppose that we have an agent $\langle \mathcal{D}, \delta^i\rangle$, and a supervision specification δ^s which are online SD. Suppose also that δ^s is online controllable with respect to $\langle \mathcal{D}, \delta^i\rangle$, and that $\mathcal{CR}(\langle \mathcal{D}, \delta^s\rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i\rangle)$. Then we have that:*

1. *$\mathcal{CR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^s\rangle)$, i.e., the complete runs of $\Sigma^w_{onl}(\delta^s, \delta^i)$ are the complete runs of δ^s.*

2. *If $\mathcal{CR}(\langle \mathcal{D}, \delta^s\rangle) \neq \emptyset$, then $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^s\rangle)$, i.e., the partial runs of $\Sigma^w_{onl}(\delta^s, \delta^i)$ are the good runs of δ^s.*

3. *If $\mathcal{CR}(\langle \mathcal{D}, \delta^s\rangle) \neq \emptyset$, then $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$, i.e., partial runs must be good runs, and the resulting program is "non blocking".*

It is also easy to show that none of these properties hold for arbitrary non-controllable programs.

Now we can show that if we apply this weak lookahead search to $\delta^i \&^{onl}_{A_u} \delta^s$, we obtain a program that has the same partial runs as $mps_{onl}(\delta^s, \sigma)$ and is thus non-blocking:

Theorem 4.7

$$\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^i \&^{onl}_{A_u} \delta^s, \delta^i)\rangle) = \mathcal{RR}(\langle \mathcal{D}, \delta^i \,\&\, mps_{onl}(\delta^s, \sigma)\rangle).$$

If we apply the weak online search construct over $\delta^i_6 \&^{onl}_{A_u} \delta^s_6$ in Example 4.4, we no longer have an online transition involving action B; the only possible online transition is $\langle \Sigma^w_{onl}(\delta^i_6 \&^{onl}_{A_u} \delta^s_6, \delta^i_6), \epsilon\rangle \rightarrow_A \langle \Sigma^w_{onl}(nil \&^{onl}_{A_u} \delta^s_6, nil), A\rangle$ where action A is performed, after which we have $(\langle \Sigma^w_{onl}(nil \&^{onl}_{A_u} \delta^s_6, nil), A\rangle)^{\checkmark}$.

4.3.5 Travel Planning Example Revisited

Let's return to the travel planning example of Section 4.1. There we presented a generic travel planning agent/process $\delta_{travelPlanner}$ and a simple customization supervision specification $\delta_{client1}$ where the client requires at least one hotel to be proposed provided one has a room available and that hotel $HtlX$ is never booked. What is the online MPS based on this supervision specification? Essentially, the supervised process cannot report failure unless it has queried all hotels at the destination other than $HtlX$ and none has a room available; if some such hotel has a room available, then one such hotel must be selected; moreover, $HtlX$ cannot be selected because if it is selected, then it must be proposed, and then the client may choose it and then it must be booked, thus violating the supervision specification.

The resulting online MPS can be obtained by inserting the following program in the $\delta_{travelPlanner}$ process right after the lines for querying/selecting airlines and/or hotels:

$$\begin{array}{l}\neg SelHtl(cID, HtlX) \land \\ (\exists htlID.\ ProposedHtl(cID, htlID) \lor \\ \ \forall htlID. IsHtl(htlID, dc) \land htlID \neq HtlX \supset \\ \ RpldHtl(cID, htlID, dC, bD, eD, hC, NotAvail)?;\end{array}$$

Observe that the intersection of the process and supervision specification $\delta_{travelPlanner}$ & $\delta_{client1}$ does not give the right result in this case. In particular, it allows the agent to select $HtlX$ and so it is not controllable for $\delta_{travelPlanner}$, since if $HtlX$ is selected, then it must be proposed, and then the client may choose it, in which case it it must be booked and the supervision specification process will not terminate successfully.

Now let's look at a second example of customization. Suppose the client has a given budget (B) and does not want the total cost of hotels (hC) and flights (aC) proposed to him to be over this budget. One could represent this specification as follows:

$$\begin{array}{l}\delta_{client2}(cID, oC, dC, bD, eD) = \\ \ \pi a.(a; \neg\exists\ htlID. ProposedHtl(cID, htlID) \land \\ \ \ RpldHtl(cID, htlID, dC, bD, eD, hC, OK) \land \\ \ \ \forall airID. ProposedAir(cID, airID) \land \\ \ \ RpldAir(cID, airID, oC, dC, bD, eD, aC, OK) \supset \\ \ \ hC + aC > B?)^{*}\end{array}$$

i.e., the process should never have proposed a hotel such that with every proposed flight the total cost of hotel and flight exceeds the budget (normally, this would be combined with other constraints). Intuitively, the online MPS for this specification is such that the agent can do anything it wants, as long as it never selects any flight, or if it does select a flight, then it does not select a hotel unless it knows of an available flight such that the combined hotel and flight cost is within budget and then such a flight must be selected before the selected proposals are displayed. Note that the agent may choose to simply report failure or avoid selecting any hotel. Again, the intersection of the agent process and supervision specification is not controllable with respect to the former, and is not the MPS. If a flight has been selected and a hotel is selected without the agent knowing of an available flight such that the combined hotel and flight cost is within budget, then the responses of airline websites about flight availability and cost, which are uncontrollable, may not yield such a flight, and the process may have no way of satisfying the constraint and completing successfully.

In a similar way, it is possible to perform configuration of the generic travel planner process for a travel service provider. For example, suppose we have a service provider for business executives that only offers 4 or 5 star hotels which are located in a downtown area. We can define a supervision specification for this configuration task as follows:

$$\begin{aligned}
&\delta_{bzProfileConfig}(cID, oC, dC, bD, eD) = \\
&\quad \pi a.(a; [\neg \exists htlID.QrdHtlWS(cID, htlID, dC, bD, eD) \wedge \\
&\quad \neg BzHtlProfile(CID, htlID))]?)^* \\
&\text{where } BzHtlProfile(cID, htlID) = \\
&\quad [HtlRating(htlID, 4) \vee HtlRating(htlID, 5)] \wedge \\
&\quad HtlArea(htlID, Downtown)
\end{aligned}$$

Personalization of the travel planner process for a client can be done simultaneously with configuration for the service provider. In this case, the supervision specification will be the intersection of the service provider's configuration specification with the the client's personalization specification.

4.4 Discussion

In this chapter, building upon DLM's proposal [35] and earlier work on supervisory control of discrete event systems [129, 166, 24], we have developed an account of supervision for agents that execute online and can acquire new information through sensing and exogenous actions as they operate.

In the area of web service composition, a highly influential approach by McIlraith and Son [113] involves customizing a generic ConGolog process based on the user's constraints and preferences. The customer's constraints with respect to an action a are captured by the fluent $Desirable(a, s)$; this is in contrast to our approach where the specification is separate from the agent program, providing flexibility and evolvability of the system. Fritz and McIlraith [61] represent customer's requirements as LTL constraints complied into ConGolog and evaluated over a finite horizon. These approaches do not consider uncontrollable actions; moreover, they both assume a middle ground between offline and online execution that depends on reasonable persistence of sensed information, which in many cases, does not hold.

In "Roman Model" approach to behavior composition, Sardina and DeGiacomo [138] synthesize a controller that orchestrates the concurrent execution of library of available (non-deterministic) ConGolog programs to realize a target program not in the library. However, they assume complete information on the initial situation, and their controller is not maximally permissive. In related work, Yadav et al. [169] consider optimal realization of the target behavior (in the presence of uncontrollable exogenous events) when its full realization is not possible. This work does not synthesize a supervisor and uses a controller. Also, it does not assume the controllability of events to be dynamic. De Giacomo et al. [44] however, synthesize a *controller generator* that represents all possible compositions of the target behavior and may adapt reactively based on runtime feedback. In a more recent work Felli et al. [58] relate the notion of a composition controller in the "Roman Model" approach to behavior composition to that of a supervisor in SCDES. Differently from our framework which uses a rich first order language, these three approaches model behaviors/services as (non-deterministic) finite state transition systems.

In the area on norm enforcement, the approach by Alechina et al. [2] regulates multiagent systems using regimented norms. A transition system describes the behavior of a (multi-) agent system and a guard function (characterized by LTL formulae with past operators) can enable/disable options that (could) violate norms after a system history (possibly using bounded lookahead). This work does not consider uncontrollable events, and is not based on expressive first order logic language. Gabaldon [65] investigates expressing and enforcing norms in the Golog programming language. Norms are encoded as additional preconditions of actions. In related work on control knowledge, Gabaldon [64] provides a procedure for compiling search control knowledge in non-Markovian action theories in the situation calculus. Control knowledge is added to precondition axioms of actions. Adding constraints as preconditions of actions in the latter two approaches, can not provide the flexibility and system evolvability that our framework can provide, as the supervision specification is encoded separately from agent behavior. Moreover, these two approaches do not consider uncontrollable actions and online executions.

Aucher [7] reformulates the results of supervisory control theory in terms of model checking problems in an epistemic temporal logic. Our work differs from this approach in that due to its first-order logic foundations, it can handle infinite object domains and infinite states. It also enables users to express the system model and the specifications in a high-level expressive language.

In the literature on supervisory control theory, non-determinism is often considered as the result of lack of information (e.g., partial observation or unmodeled internal dynamics of a system). Non-deterministic plants (e.g., [92]), non-deterministic supervisors (e.g., [87]) and non-deterministic specifications (e.g., [174]) have been studied. However, lack of information in our framework stems from incomplete knowledge in the initial state; and as the agent executes her program, she may learn new information. Our approach also differs from *limited-lookahead controller* [25, 76, 159] which uses a N-Step ahead projection (a tree) of all the possible continuations of the executed prefix to make a control decision to handle cases where there is lack of information on the possible system variations. In our framework the modeler has knowledge on possible system variations, and we use a search construct to ensure non-blocking executions.

5 Abstracting Offline Agent Behavior

Abstraction is a pervasive activity in our commonsense reasoning, as it helps solving the task at hand by identifying a *simpler* version of the task which eliminates the less important aspects of the problem [137]. For instance, when shipping items from a warehouse to a customer, we initially tend to think in terms of more abstract actions such as sending the shipment by air or via land. Then, at a more refined level, we consider more concrete actions such as the specific paths that the shipment can take to reach the customer. Such a *hierarchical* structure in reasoning seems essential in managing the complexity of many everyday tasks.

In this chapter,[28] we develop a general framework for *abstraction of offline agent behavior* based on the situation calculus [110, 132] and the ConGolog [33] agent programming language. We assume that one has a high-level/abstract action theory, a low-level/concrete action theory, and a *refinement mapping* between the two. The mapping associates each high-level primitive action to a (possibly non-deterministic) ConGolog program defined over the low-level action theory that "implements it". Moreover, it maps each high-level fluent to a state formula in the low-level language that characterizes the concrete conditions under which it holds.

In this setting, we define a notion of a high-level theory being a *sound abstraction* of a low-level theory under a given refinement mapping. The formalization involves the existence of a suitable bisimulation relation [116, 117] between models of the low-level and high-level theories. 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 thinks that a refinement of a high-level action (perhaps involving exogenous actions) could occur (i.e., its executability is satisfiable), then the high level does also. Thus, sound abstractions can be used to perform effectively several forms of reasoning about action, such as planning, agent monitoring, and generating high-level explanations of low-level behavior. We also provide a proof-theoretic characterization that gives us the basis for automatically verifying that we have a sound abstraction.

[28]The results of this chapter have already appeared in [15] and [11].

In addition, we define a dual notion of *complete abstraction* where whenever the low-level theory *entails* that some refinement of a sequence of high-level actions is executable and achieves a "translated" high-level condition, then the high level also *entails* that the action sequence is executable and the condition holds afterwards. Moreover, whenever the high level thinks that an action can occur (i.e., its executability is satisfiable), then there exists a refinement of the action that the low level thinks can happen as well.

Finally, we identify a set of basic action theory constraints that ensure that for any low-level action sequence, there is a unique high-level action sequence that it refines. This allows us to track/monitor what the low-level agent is doing and describe it in abstract terms (i.e., provide high-level explanations) e.g., to a client or manager.

5.1 Refinement Mappings

Suppose that we have a basic action theory $\mathcal{D}_l$ and another basic action theory $\mathcal{D}_h$. We would like to characterize whether $\mathcal{D}_h$ is a reasonable abstraction of $\mathcal{D}_l$. Here, we consider $\mathcal{D}_l$ as representing the *low-level* (LL) (or *concrete*) action theory/agent and $\mathcal{D}_h$ the *high-level* (HL) (or *abstract*) action theory/agent. We assume that $\mathcal{D}_h$ (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. $\mathcal{F}_l$). For simplicity, we assume that $\mathcal{D}_h$ and $\mathcal{D}_l$, share no domain specific symbols except for the set of standard names for objects $\mathcal{N}$. Moreover, for simplicity, and without loss of generality, we assume that there are no functions that return objects; object terms are only constants or variables. We also assume that there are no non-fluent predicates.

We want to relate expressions in the language of $\mathcal{D}_h$ and expressions in the language of $\mathcal{D}_l$. We say that a function m is a *refinement mapping* from $\mathcal{D}_h$ to $\mathcal{D}_l$ if and only if:

1. for every high-level primitive action type A in $\mathcal{A}_h$, $m(A(\vec{x})) = \delta_A(\vec{x})$, where $\delta_A(\vec{x})$ is a ConGolog program over the low-level theory $\mathcal{D}_l$ whose only free variables are $\vec{x}$, the parameters of the high-level action type; intuitively, $\delta_A(\vec{x})$ represents how the high-level action $A(\vec{x})$ can be implemented at the low level; since we use programs to specify the action sequences the agent may perform, we require that $\delta_A(\vec{x})$ be situation-determined, i.e., the remaining program is always uniquely determined by the situation [35];

2. for every high-level primitive fluent $F(\vec{x})$ (situation-suppressed) in $\mathcal{F}_h$, $m(F(\vec{x})) = \phi_F(\vec{x})$, where $\phi_F(\vec{x})$ is a situation-suppressed formula over the language of $\mathcal{D}_l$, and the only free variables are $\vec{x}$, the object parameters of the high-level fluent; intuitively $\phi_F(\vec{x})$ represents the *low-level condition* under which $F(\vec{x})$ holds in a situation.

Note that we can map a fluent in the high-level theory to a fluent in the low-level theory, i.e., $m(F_h(\vec{x})) = F_l(\vec{x})$, which effectively amounts to having the low-level fluent be present in the high-level theory. Similarly, one can include low-level actions in the high-level theory.

Example 5.1 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 5.1 (warehouse and cafe images are from freedesignfile.com).

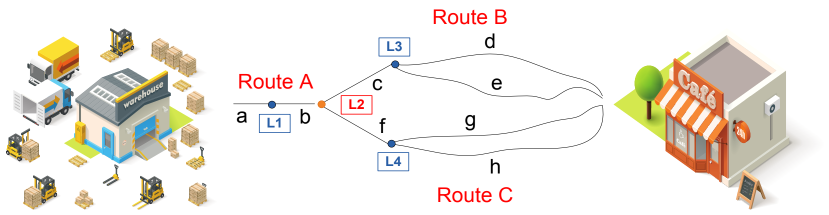

Figure 5.1: Transport Logistics Example

High-Level BAT $\mathcal{D}_h^{eg}$ At the high level, we abstract over navigation and delivery procedure details. We have actions that represent choices of major routes ($takeRoute(sID, r, o, d)$) and delivering a shipment ($deliver(sID)$). We have the following fluents: $At_{HL}(sID, l, s)$ which indicates the shipment with ID sID is at location l at situation s, $CnRoute_{HL}(r, o, d, s)$ which states route r connects location o to d at situation s, $Priority(sID, s)$ that indicates shipment sID is of type priority at s, $Dest_{HL}(sID, l, s)$ that designates the destination of the shipment sID to be location l at s, and $Delivered(sID, s)$ which indicates shipment sID has been delivered at s.

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

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

The action $takeRoute(sID, r, o, d)$ can be performed to take shipment with ID sID from origin location o to destination location d via route r (see Figure 5.1), and is executable when the shipment is initially at o and route r connects o to d; moreover, priority shipments cannot be sent by route Rt_B (note that we refer to route X in Figure 5.1 as Rt_X). Action $deliver(sID)$ can be performed to deliver shipment sID and is executable when sID is at its destination.

The high-level BAT also includes the following successor state axioms:

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

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

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

$$\begin{aligned}
&\mathit{Dest}_{HL}(123, \mathit{Cf}, S_0), \quad \mathit{At}_{HL}(123, W, S_0),\\
&\mathit{CnRoute}_{HL}(\mathit{Rt}_A, W, L2, S_0), \quad \mathit{CnRoute}_{HL}(\mathit{Rt}_B, L2, \mathit{Cf}, S_0), \mathit{CnRoute}_{HL}(\mathit{Rt}_C, L2, \mathit{Cf}, S_0)
\end{aligned}$$

Note that it is not known whether 123 is a priority shipment.

Low Level BAT $\mathcal{D}_l^{eg}$ At the low level, we model navigation and delivery in a more detailed way. The agent has a more detailed map with more locations and roads between them. He also takes road closures into account. Performing delivery involves unloading the shipment and getting a signature. We have the following fluents: $\mathit{CnRoad}(t, o, d, s)$ which states road t connects location o to d at situation s, $\mathit{Closed}(t, s)$ that indicates road t is closed at s, $\mathit{Express}(\mathit{sID}, s)$ which states shipment sID is of type express at s, $BadWeather(s)$ which indicates we have bad weather at s, $\mathit{Unloaded}(\mathit{sID}, s)$ that states shipment sID is unloaded at s, and $\mathit{Signed}(\mathit{sID}, s)$ which indicates shipment sID has been signed for at s. The fluent $\mathit{At}_{LL}(\mathit{sID}, l, s)$ is defined similarly to $\mathit{At}_{HL}(\mathit{sID}, l, s)$. Fluents $\mathit{CnRoute}_{LL}(r, o, d, s)$ and $\mathit{Dest}_{LL}(\mathit{sID}, l, s)$ in the low-level theory are defined similarly to $\mathit{CnRoute}_{HL}(r, o, d, s)$ and $\mathit{Dest}_{HL}(\mathit{sID}, l, s)$ respectively.

The low-level BAT $\mathcal{D}_l^{eg}$ includes the following action precondition axioms:

$$\begin{aligned}
&\mathit{Poss}(\mathit{takeRoad}(\mathit{sID}, t, o, d), s) \equiv o \neq d \wedge\\
&\quad \mathit{At}_{LL}(\mathit{sID}, o, s) \wedge \mathit{CnRoad}(t, o, d, s) \wedge \neg \mathit{Closed}(t, s) \wedge\\
&\quad (d = L3 \supset \neg(BadWeather(s) \vee \mathit{Express}(\mathit{sID}, s)))\\
&\mathit{Poss}(\mathit{unload}(\mathit{sID}), s) \equiv \exists l. \mathit{Dest}_{LL}(\mathit{sID}, l, s) \ \wedge \mathit{At}_{LL}(\mathit{sID}, l, s)\\
&\mathit{Poss}(getSignature(\mathit{sID}), s) \equiv \mathit{Unloaded}(\mathit{sID}, s)
\end{aligned}$$

Thus, the action $\mathit{takeRoad}(\mathit{sID}, t, o, d)$, where the agent takes shipment sID from origin location o to destination d via road t, is executable provided that t connects o to d, sID is at o, and t is not closed; moreover, a road to $L3$ cannot be taken if the weather is bad or sID is an express shipment as it would likely violate quality of service requirements. The action $\mathit{unload}(\mathit{sID})$ can be performed to unload the shipment sID and is possible when the shipment has reached its destination. Finally, after the shipment is unloaded the action $getSignature(\mathit{sID})$ can be performed.

The low-level BAT includes the following successor state axioms:

$$\begin{aligned}
&\mathit{Unloaded}(\mathit{sID}, do(a, s)) \equiv a = \mathit{unload}(\mathit{sID}) \vee \mathit{Unloaded}(\mathit{sID}, s)\\
&\mathit{Signed}(\mathit{sID}, do(a, s)) \equiv a = getSignature(\mathit{sID}) \vee \mathit{Signed}(\mathit{sID}, s)
\end{aligned}$$

The sucessor state axiom for At_{LL} is like the one for At_{HL} with *takeRoute* replaced by *takeRoad*. For the other fluents, we have successor state axioms specifying that they are unaffected by any actions. Note that we could easily include exogenous actions for road closures and change in weather, new shipment orders, etc.

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

$$\neg BadWeather(S_0),\ Closed(r, S_0) \equiv r = Rd_e,$$
$$Express(123, S_0),\ Dest_{LL}(123, Cf, S_0),\ At_{LL}(123, W, S_0)$$

together with a complete specification of *CnRoad* and $CnRoute_{LL}$. We refer to road x in Figure 5.1 as Rd_x.

Refinement Mapping m^{eg} We specify the relationship between the high-level and low-level BATs through the following refinement mapping m^{eg}:

$$\begin{aligned}
&m^{eg}(takeRoute(sID, r, o, d)) = \\
&\quad (r = Rt_A \wedge CnRoute_{LL}(Rt_A, o, d))?; \\
&\quad\ \pi t.takeRoad(sID, t, o, L1); \pi t'.takeRoad(sID, t', L1, d) \mid \\
&\quad (r = Rt_B \wedge CnRoute_{LL}(Rt_B, o, d))?; \\
&\quad\ \pi t.takeRoad(sID, t, o, L3); \pi t'.takeRoad(sID, t', L3, d) \mid \\
&\quad (r = Rt_C \wedge CnRoute_{LL}(Rt_C, o, d))?; \\
&\quad\ \pi t.takeRoad(sID, t, o, L4); \pi t'.takeRoad(sID, t', L4, d)
\end{aligned}$$

$$\begin{aligned}
&m^{eg}(deliver(sID)) = unload(sID); getSignature(sID) \\
&m^{eg}(Priority(sID)) = BadWeather \vee Express(sID) \\
&m^{eg}(Delivered(sID)) = Unloaded(sID) \wedge Signed(sID) \\
&m^{eg}(At_{HL}(sID, loc)) = At_{LL}(sID, loc) \\
&m^{eg}(CnRoute_{HL}(r, o, d)) = CnRoute_{LL}(r, o, d) \\
&m^{eg}(Dest_{HL}(sID, l)) = Dest_{LL}(sID, l)
\end{aligned}$$

Thus, taking route Rt_A involves first taking a road from the origin o to $L1$ and then taking another road from $L1$ to the destination d. For the other two routes, the refinement mapping is similar except a different intermediate location must be reached. Note that we could easily write programs to specify refinements for more complex routes, e.g., that take a sequence of roads from o to d going through intermediate locations belonging to a given set. We refine the high-level fluent $Priority(sID)$ to the condition where either the weather is bad or the shipment is express. $\square$

5.2 m-Bisimulation

To relate high-level and low-level models/theories, we resort to a suitable notion of bisimulation. Let M_h be a model of the high-level BAT $\mathcal{D}_h$, M_l a model of the low-level BAT $\mathcal{D}_l \cup \mathcal{C}$, and m a refinement mapping from $\mathcal{D}_h$ to $\mathcal{D}_l$.

We first define a local condition for the bisimulation. We say that situation s_h in M_h is m-isomorphic to situation s_l in M_l, written $s_h \sim_m^{M_h,M_l} s_l$, if and only if

> $M_h, v[s/s_h] \models F(\vec{x}, s)$ iff $M_l, v[s/s_l] \models m(F(\vec{x}))[s]$ for every high-level primitive fluent $F(\vec{x})$ in $\mathcal{F}_h$ and every variable assignment v ($v[x/e]$ stands for the assignment that is like v except that x is mapped to e).

A relation $B \subseteq \Delta_S^{M_h} \times \Delta_S^{M_l}$ (where Δ_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:

1. $s_h \sim_m^{M_h,M_l} s_l$, i.e., s_h in M_h is m-isomorphic to situation s_l in M_l;

2. for every high-level primitive action type A in $\mathcal{A}_h$, if there exists s'_h such that $M_h, v[s/s_h, s'/s'_h] \models Poss(A(\vec{x}), s) \land s' = do(A(\vec{x}), s)$, then there exists s'_l such that $M_l, v[s/s_l, s'/s'_l] \models Do(m(A(\vec{x})), s, s')$ and $\langle s'_h, s'_l\rangle \in B$;

3. for every high-level primitive action type A in $\mathcal{A}_h$, if there exists s'_l such that $M_l, v[s/s_l, s'/s'_l] \models Do(m(A(\vec{x})), s, s')$, then there exists s'_h such that $M_h, v[s/s_h, s'/s'_h] \models Poss(A(\vec{x}), s) \land s' = do(A(\vec{x}), s)$ and $\langle s'_h, s'_l\rangle \in B$.

We say that M_h *is bisimilar to M_l relative to refinement mapping m, 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 $\langle S_0^{M_h}, S_0^{M_l}\rangle \in B$.

Given these definitions, we immediately get the following results. First, we can show that m-isomorphic situations satisfy the same high-level situation-suppressed formulas (for proofs of our results in this chapter, see Appendix A.2):

Lemma 5.1 *If $s_h \sim_m^{M_h,M_l} s_l$, then for any high-level situation-suppressed formula ϕ, we have that:*

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

Note that $m(\phi)$ stands for the result of substituting every fluent $F(\vec{x})$ in situation-suppressed formula ϕ by $m(F(\vec{x}))$.

Moreover, it is straightforward to show that in m-bisimilar models, the same sequences of high-level actions are executable, and that in the resulting situations, the same high-level situation-suppressed formulas hold:

Theorem 5.2 *If $M_h \sim_m M_l$, then for any sequence of ground high-level actions $\vec{\alpha}$ and any high-level situation-suppressed formula ϕ, we have that*

$$M_l \models \exists s' Do(m(\vec{\alpha}), S_0, s') \wedge m(\phi)[s'] \quad \textit{if and only if} \quad M_h \models \mathit{Executable}(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)].$$

Note that $m(\alpha_1, \ldots, \alpha_n) \doteq m(\alpha_1); \ldots ; m(\alpha_n)$ for $n \geq 1$ and $m(\epsilon) \doteq nil$.

5.3 Sound Abstraction

To ensure that the high-level theory is consistent with the low-level theory and mapping m, we require that for every model of the low-level theory, there is an m-bisimilar structure that is a model of the high-level theory.

We say that $\mathcal{D}_h$ is a *sound abstraction of $\mathcal{D}_l$ relative to refinement mapping m* if and only if, for all models M_l of $\mathcal{D}_l \cup \mathcal{C}$, there exists a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$.

Example 5.2 Returning to Example 5.1, it is straightforward to show that it involves a high-level theory $\mathcal{D}_h^{eg}$ that is a sound abstraction of the low-level theory $\mathcal{D}_l^{eg}$ relative to the mapping m^{eg}. We discuss how we prove this later. □

Sound abstractions have many interesting and useful properties. First, from the definition of sound abstraction and Theorem 5.2, we immediately get the following result:

Corollary 5.3 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any sequence of ground high-level actions $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, if $\mathcal{D}_l \cup \mathcal{C} \cup \{\exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]\}$ is satisfiable, then $\mathcal{D}_h \cup \{\mathit{Executable}(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]\}$ is also satisfiable. In particular, if $\mathcal{D}_l \cup \mathcal{C} \cup \{\exists s. Do(m(\vec{\alpha}), S_0, s)\}$ is satisfiable, then $\mathcal{D}_h \cup \{\mathit{Executable}(do(\vec{\alpha}, S_0))\}$ is also satisfiable.*

Thus if the low-level agent/theory thinks that a refinement of $\vec{\alpha}$ (perhaps involving exogenous actions) may occur (with $m(\phi)$ holding afterwards), the high-level agent/theory also thinks that $\vec{\alpha}$ may occur (with ϕ holding afterwards). If such a refinement actually occurs it will thus be consistent with the high-level theory.

We can also show that if the high-level theory entails that some sequence of high-level actions $\vec{\alpha}$ is executable, and that in the resulting situation, a situation-suppressed formula ϕ holds, then the low-level theory must also entail that some refinement of $\vec{\alpha}$ is executable and that in the resulting situation $m(\phi)$ holds:

Theorem 5.4 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any ground high-level action sequence $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, if $\mathcal{D}_h \models \mathit{Executable}(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$, then $\mathcal{D}_l \cup \mathcal{C} \models \exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$.*

We can immediately relate the above result to *planning*. In the situation calculus, the planning problem is usually defined as follows [132]:

> Given a BAT $\mathcal{D}$, and a situation-suppressed goal formula ϕ, find a ground action sequence $\vec{a}$ such that $\mathcal{D} \models \mathit{Executable}(do(\vec{a}, S_0)) \wedge \phi[do(\vec{a}, S_0)]$.

Thus, Theorem 5.4 means that if we can find a plan $\vec{\alpha}$ to achieve a goal ϕ at the high level, i.e., $\mathcal{D}_h \models \mathit{Executable}(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$, then it follows that there exists a refinement of $\vec{\alpha}$ that achieves ϕ at the low level, i.e., $\mathcal{D}_l \cup \mathcal{C} \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$. However, note that the refinement could in general be different from model to model. But if, in addition, we have complete information at the low level, i.e., a single model for $\mathcal{D}_l$, then, since we have standard names for objects and actions, we can always obtain a plan to achieve the goal ϕ by finding a refinement in this way, i.e., there exists a ground low-level action sequence $\vec{a}$ such that $\mathcal{D}_l \cup \mathcal{C} \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, s)) \wedge m(\phi)[do(\vec{a}, s)]$. The search space of refinements of $\vec{\alpha}$ would typically be much smaller than the space of all low-level action sequences, thus yielding important efficiency benefits.

We can also show that if $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ with respect to a mapping, then the different sequences of low-level actions that are refinements of a given high-level primitive action sequence all have the same effects on the high-level fluents, and more generally on high-level situation-suppressed formulas, i.e., from the high-level perspective they are deterministic:

Corollary 5.5 *If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, then for any sequence of ground high-level actions $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, we have that*

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s \forall s'.Do(m(\vec{\alpha}), S_0, s) \wedge Do(m(\vec{\alpha}), S_0, s') \supset (m(\phi)[s] \equiv m(\phi)[s'])$$

An immediate consequence of the above is the following:

Corollary 5.6 *If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, then for any sequence of ground high-level actions $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, we have that*

$$\mathcal{D}_l \cup \mathcal{C} \models (\exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]) \supset (\forall s.Do(m(\vec{\alpha}), S_0, s) \supset m(\phi)[s])$$

It is also easy to show that if some refinement of the sequence of high-level actions $\vec{\alpha}\beta$ is executable, then there exists a refinement of β that is executable after executing any refinement of $\vec{\alpha}$:

Theorem 5.7 *If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, then for any sequence of ground high-level actions $\vec{\alpha}$ and for any ground high-level action β, we have that*

$$\mathcal{D}_l \cup \mathcal{C} \models \exists s.Do(m(\vec{\alpha}\beta), S_0, s) \supset (\forall s.Do(m(\vec{\alpha}), S_0, s) \supset \exists s'.Do(m(\beta), s, s'))$$

Notice that this applies to all prefixes of $\vec{\alpha}$, so using Corollary 5.6 as well, we immediately get that:

Corollary 5.8 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any ground high-level action sequence $\alpha_1, \ldots, \alpha_n$, and for any high-level situation-suppressed formula ϕ, then we have that:*

$$\begin{aligned}\mathcal{D}_l \cup \mathcal{C} \models (\exists s.Do(m(\alpha_1, \ldots, \alpha_n), S_0, s) \wedge m(\phi)[s]) \supset \\ ((\forall s.Do(m(\alpha_1, \ldots, \alpha_n), S_0, s) \supset m(\phi)[s]) \wedge (\exists s.Do(m(\alpha_1), S_0, s)) \wedge \\ \textstyle\bigwedge_{2 \leq i \leq n}(\forall s.Do(m(\alpha_1, \ldots, \alpha_{i-1}), S_0, s) \supset \exists s'.Do(m(\alpha_i), s, s')))\end{aligned}$$

These results mean that if a ground high-level action sequence achieves a high-level condition ϕ, we can choose refinements of the actions in the sequence independently and be certain to obtain a refinement of the complete sequence that achieves ϕ. We can exploit this result in planning to gain even more efficiency. If we can find a plan $\alpha_1, \ldots, \alpha_n$ to achieve a goal ϕ at the high level, then there exists a refinement of $\alpha_1, \ldots, \alpha_n$ that achieves $m(\phi)$ at the low level, and we can obtain it by finding refinements of the high-level actions α_i for $i : 1 \leq i \leq n$ one by one, without ever having to backtrack. The search space would typically be exponentially smaller in the length of the high-level plan n. If we have complete information at the low level, then we can always obtain a refined plan to achieve $m(\phi)$ in this way.

Example 5.3 Returning to our running example, we can show that the action sequence $\vec{\alpha} = [takeRoute(123, Rt_A, W, L2),\ takeRoute(123, Rt_C, L2,\ Cf), deliver(123)]$ is a valid high-level plan to achieve the goal $\phi_g = Delivered(123)$ of having delivered shipment 123, i.e., $\mathcal{D}_h^{eg} \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi_g[do(\vec{\alpha}, S_0)]$. Since $\mathcal{D}_h^{eg}$ is a sound abstraction of the low-level theory $\mathcal{D}_l^{eg}$ relative to the mapping m^{eg}, we know that we can find a refinement of the high-level plan $\vec{\alpha}$ that achieves the refinement of the goal $m^{eg}(\phi_g) = Unloaded(123) \wedge Signed(123)$. In fact, we can show that $\mathcal{D}_l^{eg} \cup \mathcal{C} \models Do(m^{eg}(\vec{\alpha}), S_0, do(\vec{a}\vec{b}\vec{c}, S_0)) \wedge m^{eg}(\phi_g)[do(\vec{a}\vec{b}\vec{c}, S_0)]$ for $\vec{a} = [takeRoad(123, Rd_a, W, L1), takeRoad(123, Rd_b,\ L1, L2)]$, $\vec{b} = [takeRoad(123, Rd_f, L2, L4),\ takeRoad(123, Rd_g, L4, Cf)]$, and $\vec{c} = [unload(123),\ getSignature(123)]$. □

Now, let us define some low-level programs that characterize the refinements of high-level action/action sequences:[29]

$$\begin{aligned}&\textsc{any1hlref} \doteq \ \{\pi\vec{x}.m(A_i(\vec{x})) \mid A_i \in \mathcal{A}_h\}, \text{ i.e., do any high-level primitive action,}\\ &\textsc{anyseqhlref} \doteq \textsc{any1hlref}^* \ \text{ i.e., do any sequence of high-level actions.}\end{aligned}$$

[29]These programs correctly characterize the *complete runs* of the agent executing the refinements of high-level actions/action sequences at the low level. However, note that ANY1HLREF is not situation-determined, as refinements of high-level actions may share prefixes. In Chapter 6, as we also need to consider *partial runs* of the agent executing such programs, we define a new **setp**() construct to obtain situation-determined programs that characterize the refinements of high-level action/action sequences.

How does one verify that one has a sound abstraction? The following yields a straightforward method for this:

Theorem 5.9 *$\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ relative to mapping m if and only if*

(a) $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa} \models m(\phi)$*, for all* $\phi \in D^h_{S_0}$*,*

(b) $\mathcal{D}^l \cup \mathcal{C} \models \forall s. Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}. (m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s')),$$

(c) $\mathcal{D}^l \cup \mathcal{C} \models \forall s. Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'. (Do(m(A_i(\vec{x})), s, s') \supset$$
$$\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y} (m(\phi^{ssa}_{F_i, A_i}(\vec{y}, \vec{x}))[s] \equiv m(F_i(\vec{y}))[s'])),$$

where $\phi^{Poss}_{A_i}(\vec{x})$ is the right hand side (RHS) of the precondition axiom for action $A_i(\vec{x})$, and $\phi^{ssa}_{F_i, A_i}(\vec{y}, \vec{x})$ is the RHS of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using $\mathcal{D}^h_{ca}$.

The above essentially gives us a "proof theoretic" characterization of sound abstraction, in the sense that it allows us to show that we have a sound abstraction by proving that a number of properties are entailed by the low-level theory. Conditions (a), (b), and (c) are all properties of programs that standard verification techniques can deal with. The theorem also means that if $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ with respect to m, then $\mathcal{D}^l$ must entail the mapped high-level successor state axioms and entail that the mapped conditions for a high-level action to be executable (from the precondition axioms of $\mathcal{D}^h$) correctly capture the executability conditions of their refinements (these conditions must hold after any sequence of refinements of high-level actions, i.e., in any situation s where $Do(\text{ANYSEQHLREF}, S_0, s)$ holds).

Example 5.4 Returning to our running example, it is straightforward to show that it involves a high-level theory $\mathcal{D}^{eg}_h$ that is a sound abstraction of the low-level theory $\mathcal{D}^{eg}_l$ relative to the mapping m^{eg}. $\mathcal{D}^{S_0}_l$ entails the "translation" of all the facts about the high-level fluents $CnRoute_{HL}$, $Dest_{HL}$ and At_{HL} that are in $\mathcal{D}^{S_0}_h$. Moreover, $\mathcal{D}^{eg}_l$ entails that the mapping of the preconditions of the high-level actions *deliver* and *takeRoute* correctly capture the executability conditions of their refinements. $\mathcal{D}^{eg}_l$ also entails the mapped high-level successor state axiom for fluent At_{HL} and action *takeRoute* and similarly for *Delivered* and action *deliver* (other actions have no effects). Thus, $\mathcal{D}^{eg}_h$ is a sound abstraction of $\mathcal{D}^{eg}_l$ relative to m^{eg}. □

5.4 Complete Abstraction

When we have a sound abstraction $\mathcal{D}_h$ of a low-level theory $\mathcal{D}_l$ with respect to a mapping m, the high-level theory $\mathcal{D}_h$'s conclusions are always sound with respect to the more refined theory $\mathcal{D}_l$, but $\mathcal{D}_h$ may have less information than $\mathcal{D}_l$ regarding high-level actions and conditions. $\mathcal{D}_h$ may consider it possible that a high-level action sequence is executable (and achieves a goal) when $\mathcal{D}_l$ knows it is not. The low-level theory may know/entail that a refinement of a high-level action sequence achieves a goal without the high level knowing/entailing it. We can define a stronger notion of abstraction that ensures that the high-level theory knows everything that the low-level theory knows about high-level actions and conditions.

We say that $\mathcal{D}_h$ is a *complete abstraction of $\mathcal{D}_l$ relative to refinement mapping m* if and only if, for all models M_h of $\mathcal{D}_h$, there exists a model M_l of $\mathcal{D}_l \cup \mathcal{C}$ such that $M_l \sim_m M_h$.

From the definition of complete abstraction and Theorem 5.2, we immediately get the following converses of Corollary 5.3 and Theorem 5.4:

Corollary 5.10 *Suppose that $\mathcal{D}_h$ is a complete abstraction of $\mathcal{D}_l$ relative to m. Then for any sequence of ground high-level actions $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, if $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]\}$ is satisfiable, then $\mathcal{D}_l \cup \mathcal{C} \cup \{\exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]\}$ is satisfiable. In particular, if $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is satisfiable, $\mathcal{D}_l \cup \mathcal{C} \cup \{\exists s. Do(m(\vec{\alpha}), S_0, s)\}$ is satisfiable.*

Theorem 5.11 *Suppose that $\mathcal{D}_h$ is a complete abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any ground high-level action sequence $\vec{\alpha}$ and any high-level situation-suppressed formula ϕ, if $\mathcal{D}_l \cup \mathcal{C} \models \exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$, then $\mathcal{D}_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$.*

Thus when we have a high-level theory $\mathcal{D}_h$ that is a complete abstraction of a low-level theory $\mathcal{D}_l$ with respect to a mapping m, if $\mathcal{D}_l$ knows/entails that some refinement of a high-level action sequence $\vec{\alpha}$ achieves a high-level goal ϕ, then $\mathcal{D}_h$ knows/entails that $\vec{\alpha}$ achieves ϕ, i.e., we can find all high-level plans to achieve high-level goals using $\mathcal{D}_h$.

Note that with complete abstraction alone, we don't get Corollary 5.5, as $\mathcal{D}_l \cup \mathcal{C}$ may have models that are not m-bisimilar to any model of $\mathcal{D}_h$ and where different refinements of a high-level action yield different truth-values for $m(F)$, for some high-level fluent F.

We also say that $\mathcal{D}_h$ is a *sound and complete abstraction of $\mathcal{D}_l$ relative to refinement mapping m* if and only if $\mathcal{D}_h$ is both a sound and a complete abstraction of $\mathcal{D}_l$ relative to m.

Example 5.5 Returning to our running example, the high-level theory does not know whether shipment 123 is high priority, i.e., $\mathcal{D}_h^{eg} \not\models Priority(123)[S_0]$ and $\mathcal{D}_h^{eg} \not\models \neg Priority(123)[S_0]$, but the low-level theory knows that it is, i.e., $\mathcal{D}_l^{eg} \models m^{eg}(Priority(123))[S_0]$. Thus $\mathcal{D}_h^{eg}$ has a model where $\neg Priority(123)[S_0]$ holds that is not m^{eg}-bisimilar to any model of $\mathcal{D}_l^{eg}$, and thus $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_l^{eg}$ with respect to m^{eg}, but not a complete abstraction. For instance, the high-level theory considers it possible that the shipment can be delivered by taking route A and then route B, i.e., $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge \phi_g[do(\vec{\alpha}, S_0)]\}$ is satisfiable for $\vec{\alpha} = [takeRoute(123, Rt_A, W, L2), takeRoute(123, Rt_B, L2, Cf), deliver(123)]$ and $\phi_g = Delivered(123)$.

But the low-level theory knows that $\vec{\alpha}$ cannot be refined to an executable low-level plan, i.e., $\mathcal{D}_l^{eg} \cup \mathcal{C} \models \neg\exists s.Do(m^{eg}(\vec{\alpha}), S_0, s)$. If we add $Priority(123)[S_0]$ and a complete specification of $CnRoute_{HL}$ to $\mathcal{D}_h^{eg}$, then it becomes a sound and complete abstraction of $\mathcal{D}_l^{eg}$ with respect to m^{eg}. The plan $\vec{\alpha}$ is now ruled out as $\mathcal{D}_h^{eg} \cup \{Priority(123,\, S_0)\} \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is not satisfiable. □

The following results characterize complete abstractions:

Theorem 5.12 *If $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ relative to mapping m, then $\mathcal{D}^h$ is also a complete abstraction of $\mathcal{D}^l$ with respect to mapping m if and only if for every model M_h of $\mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$, there exists a model M_l of $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa}$ such that $S_0^{M_h} \sim_m^{M_h, M_l} S_0^{M_l}$.*

Theorem 5.13 *$\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ relative to mapping m iff for every model M_h of $\mathcal{D}^h$, there exists a model M_l of $\mathcal{D}^l \cup \mathcal{C}$ such that $S_0^{M_h} \sim_m^{M_h, M_l} S_0^{M_l}$ and*

$M_l \models \forall s.Do(\text{ANYSEQHLREF}, S_0, s) \supset$

$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$

and $M_l \models \forall s.Do(\text{ANYSEQHLREF}, S_0, s) \supset$

$\qquad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$

$\qquad\quad \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i, A_i}(\vec{y}, \vec{x}))[s] \equiv m(F_i(\vec{y}))[s'])),$

where $\phi^{Poss}_{A_i}(\vec{x})$ and $\phi^{ssa}_{F_i, A_i}(\vec{y}, \vec{x})$ are as in Theorem 5.9.

5.5 Monitoring and Explanation

A refinement mapping m from a high-level action theory $\mathcal{D}_h$ to a low-level action theory $\mathcal{D}_l$ tells us what are the refinements of high-level actions into executions of low-level programs. In some application contexts, one is interested in tracking/monitoring what the low-level agent is doing and describing it in abstract terms, e.g., to a client or manager. If we have a ground low-level situation term S_l such that $\mathcal{D}_l \cup \{Executable(S_l)\}$ is satisfiable, and $\mathcal{D}_l \cup \{Do(m(\vec{\alpha}), S_0, S_l)\}$ is satisfiable, then the ground high-level action sequence $\vec{\alpha}$ is a possible way of describing in abstract terms what has occurred in getting to situation S_l. If $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is also satisfiable (it must be if $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ with respect to m), then one can also answer high-level queries about what may hold in the resulting situation, i.e., whether $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge \phi(do(\vec{\alpha}, S_0))\}$ is satisfiable, and what must hold in such a resulting situation, i.e., whether $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models \phi(do(\vec{\alpha}, S_0))$. One can also answer queries about what high-level action β might occur next, i.e., whether $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}\beta, S_0))\}$ is satisfiable.

In general, there may be several distinct ground high-level action sequences $\vec{\alpha}$ that match a ground low-level situation term S_l; even if we have complete information and a single model M_l of $\mathcal{D}_l \cup \mathcal{C}$, there may be several distinct $\vec{\alpha}$'s such that $M_l \models Do(m(\vec{\alpha}), S_0, S_l)$. For example, suppose that we have two high level actions A and B with $m(A) = (C \mid D)$ and $m(B) = (D \mid E)$. Then the low-level situation $do(D, S_0)$ is a refinement of both A and B (assuming all actions are always executable).

In many contexts, this would be counterintuitive and we would like to be able to map a sequence of low-level actions performed by the low-level agent back into a *unique* abstract high-level action sequence it refines, i.e., we would like to define an inverse mapping *function* m^{-1}. Let's see how we can do this. First, we introduce the abbreviation $lp_m(s, s')$, meaning that s' is a largest prefix of s that can be produced by executing a sequence of high-level actions:

$$\begin{aligned} lp_m(s, s') \doteq\ & Do(\text{ANYSEQHLREF}, S_0, s') \land s' \leq s \land \\ & \forall s''.(s' < s'' \leq s \supset \neg Do(\text{ANYSEQHLREF}, S_0, s'')) \end{aligned}$$

We can show that the relation $lp_m(s, s')$ is actually a total function:

Theorem 5.14 *For any refinement mapping m from $\mathcal{D}_h$ to $\mathcal{D}_l$, we have that:*

1. $D_l \cup \mathcal{C} \models \forall s.\exists s'.lp_m(s, s')$,

2. $D_l \cup \mathcal{C} \models \forall s \forall s_1 \forall s_2.lp_m(s, s_1) \land lp_m(s, s_2) \supset s_1 = s_2$.

Given this result, we can introduce the notation $lp_m(s) = s'$ to stand for $lp_m(s, s')$.

To be able to map a low-level action sequence back to a unique high-level action sequence that produced it, we need the following assumptions:

Assumption 5.1 *For any distinct ground high-level action terms α and α' we have that:*

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

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

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

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. Together, these three conditions ensure that if we have a low-level action sequence that can be produced by executing some high-level action sequence, there is a unique high-level action sequence that can produce it:

Theorem 5.15 *Suppose that we have a refinement mapping m from $\mathcal{D}_h$ to $\mathcal{D}_l$ and that Assumption 5.1 holds. Let M_l be a model of $\mathcal{D}_l \cup \mathcal{C}$. Then for any ground situation terms S_s and S_e such that $M_l \models Do(\text{ANYSEQHLREF}, S_s, S_e)$, there exists a unique ground high-level action sequence $\vec{\alpha}$ such that $M_l \models Do(m(\vec{\alpha}), S_s, S_e)$.*

Since in any model M_l of $\mathcal{D}_l \cup \mathcal{C}$, for any ground situation term S, there is a unique largest prefix of S that can be produced by executing a sequence of high-level actions, $S' = lp_m(S)$, and for any such S', there is a unique ground high-level action sequence $\vec{\alpha}$ that can produce it, we can view $\vec{\alpha}$ as the value of the inverse mapping m^{-1} for S in M_l. For this, let us introduce the following notation:

$$m^{-1}_{M_l}(S) = \vec{\alpha} \doteq M_l \models \exists s'.lp_m(S) = s' \wedge Do(m(\vec{\alpha}), S_0, s')$$

where m is a refinement mapping from $\mathcal{D}_h$ to $\mathcal{D}_l$ and Assumption 5.1 holds, M_l is a model of $\mathcal{D}_l \cup \mathcal{C}$, S is a ground low-level situation term, and $\vec{\alpha}$ is a ground high-level action sequence.

Assumption 5.1 however does not ensure that any low-level situation S can in fact be generated by executing a refinement of some high-level action sequence; if it cannot, then the inverse mapping will not return a complete matching high-level action sequence (e.g., we might have $m^{-1}_{M_l}(S) = \epsilon$). We can introduce an additional assumption that rules such cases out:[30]

Assumption 5.2

$$D_l \cup \mathcal{C} \models \forall s.Executable(s) \supset \exists \delta.Trans^*(\text{ANYSEQHLREF}, S_0, \delta, s)$$

With this additional assumption, we can show that for any executable low-level situation s, what remains after the largest prefix that can be produced by executing a sequence of high-level actions, i.e., the actions in the interval between s' and s where $lp_m(s, s')$, can be generated by some (not yet complete) refinement of a high-level primitive action:

Theorem 5.16 *If m is a refinement mapping from $\mathcal{D}_h$ to $\mathcal{D}_l$ and Assumption 5.2 holds, then we have that:*

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s, s'.Executable(s) \wedge lp_m(s, s') \supset \exists \delta.Trans^* \text{ANY1HLREF}, s', \delta, s)$$

Example 5.6 Going back to Example 5.1, assume that we have complete information at the low level and a single model M_l of $\mathcal{D}^{eg}_l$, and suppose that the sequence of (executable) low-level actions $\vec{a} = [takeRoad(123, Rd_a, W, L1), takeRoad(123, Rd_b, L1, L2)]$ has occurred. The inverse mapping allows us to conclude that the high-level action $\alpha = takeRoute(123, Rt_A, W, L2)$ has occurred, since $m^{-1}_{M_l}(do(\vec{a}, S_0)) = \alpha$.[31] Since $\mathcal{D}_h \models At_{HL}(123, L2, do(\alpha, S_0))$, we can also conclude that shipment 123 is now at location

[30]One might prefer a weaker version of Assumption 5.2. For instance, one could write a program specifying the low level agent's possible behaviors and require that situations reachable by executing this program can be generated by executing a refinement of some high-level action sequence. We discuss the use of programs to specify possible agent behaviors in the next section.

[31]If we do not have complete information at the low level, $m^{-1}_{M}(\vec{a})$ may be different for different models M of $\mathcal{D}_l$. To do high level tracking/monitoring in such cases, we need to consider all the possible mappings or impose additional restrictions to ensure that there is a unique mapping. We leave this problem for future work.

$L2$. As well, since $\mathcal{D}_h \cup \{Poss(takeRoute(123, Rt_B, L2, Cf), do(\alpha, S_0))\}$ is satisfiable, we can conclude that high-level action $takeRoute(123, Rt_B, L2, Cf)$ might occur next. Analogously, we can also conclude that high-level action $takeRoute(123, Rt_C, L2, Cf)$ might occur next. $\square$

5.6 Discussion

In this chapter, we proposed a general framework for agent abstraction based on the situation calculus and ConGolog. For simplicity, we focused on a single layer of abstraction, but the framework supports extending the hierarchy to more levels. Our approach can also support the use of ConGolog programs to specify the possible behaviors of the agent at both the high and low level, as we can follow [37] and "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.

In AI, Giunchiglia and Walsh [74] formalize abstraction as *syntactic* mappings between formulas of a concrete and a more abstract representation. Their notion of TD and TI abstractions seems similar to our notions of sound and complete abstractions respectively; however, while they focus on only syntactic aspect of abstractions that does not explicitly capture the underlying justification that lead to the abstraction, we consider bisimilar models of the theory. Nayak and Levy [122] present a *semantic* theory of abstraction. Their notion of MI and the strongest MI abstractions seems similar to our notions of sound and sound & complete abstraction. In contrast to our approach, no separate discussion on complete abstractions is provided. Moreover, no verification method is suggested; and the resolution-based procedure for constructing the strongest MI abstraction includes a number of simplifying assumptions, such as the abstract language not including equality. Both of the the above approaches formalize abstraction of *static* logical theories, while our work focuses on abstraction of *dynamic* domains.

In planning, several notions of abstraction have been investigated. One approach is *precondition elimination abstraction*, first introduced in context of ABSTRIPS [135]. This work does not consider abstraction of actions. Another approach proposes *Hierarchical Task Networks* (HTNs) (e.g., [57]), which abstract over a set of (non-primitive) tasks. Encodings of HTNs in ConGolog with enhanced features like exogenous actions and online executions have also been studied by Gabaldon [63]. In contrast to our approach, [63] uses a single BAT; also it does not provide abstraction for fluents. Planning with *macro operators* (e.g., [89]), is another approach to abstraction which represents meta-actions built from a sequence of action steps. McIlraith and Fadel [112] and Baier and McIlraith [10] investigate planning with *complex actions* (a form of macro actions) specified as Golog [103] programs. Differently from our approach, [112, 10] compile the abstracted actions into a new BAT that contains both the original and abstracted actions. Also, they only deal with deterministic complex actions and do not provide abstraction for fluents. Moreover, our approach provides a refinement mapping (similar to that of Global-As-View in data integration [97]) between an abstract BAT and a concrete BAT. Aguas et al. [1] proposed hierarchical finite state controllers for generalized planning. This work differs from our approach in that it is focused on providing a solution for a problem class that can be used to solve any particular *instance* of the class; also the framework is not based on first order logic. Another important difference between our work and the approaches focused on abstraction in planning is that they focus on improving the efficiency of planning, while our work provides a generic framework which can have applications in many areas.

6 Hierarchical Agent Supervision

As agent-based systems are becoming increasingly important in modeling and implementing complex software systems, an agent's behavior in such systems as well as her interactions with the environment tend to become more involved. Due to complexity of the behavior logic, designing and enforcing specifications for control/customization of agent's behavior can be difficult. To facilitate supervision, it is possible to consider hierarchical models where a high level abstracts over low-level behavior details.

In this chapter, building on the notion of abstraction in situation calculus action theories presented in Chapter 5 and inspired by the hierarchical supervisory control of discrete event systems [166] and the DLM framework for agent supervision, we study *hierarchical agent supervision* in the context of the situation calculus and the ConGolog agent programming language. As in Chapter 5, 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, we assume that 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. We also assume that 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 maximally permissive supervisor (MPS) for the low-level agent and supervision 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. Moreover we show that we can obtain the low-level

MPS incrementally using the high-level MPS as a guide.

6.1 Preliminaries

6.1.1 Notation

New Notation for Controllability. Notice that the definition of controllability in Section 3.3 can be re-written to avoid quantification over action sequences as follows:

$$
\begin{aligned}
&\mathit{Controllable}(\delta^s, \delta^i, s) \doteq \\
&\forall s', a_u.(\exists s''.Do(\delta^s, s, s'') \land s \leq s' \leq s'') \land A_u(a_u, s') \land \\
&\qquad (\exists s''.Do(\delta^i, s, s'') \land s < do(a_u, s') \leq s'') \supset (\exists s''.Do(\delta^s, s, s'') \land s < do(a_u, s') \leq s'')
\end{aligned}
$$

i.e., if we take an action sequence $\vec{a}$ that is a prefix of a complete run of δ^s and append to it an uncontrollable action a_u such that $\vec{a}a_u$ is a prefix of a complete run of δ^i, then $\vec{a}a_u$ must also be a prefix of a complete run of δ^s. Note that the new notation has exactly the same meaning as that of (3.23). We will also write $\mathit{Controllable}_M(\delta^s, \delta^i, s)$ as an abbreviation for $M \models \mathit{Controllable}(\delta^s, \delta^i, s)$, where M is a model (of the BAT).

m-Refinement. We say that *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 $M_h \models \mathit{Executable}(do(\vec{\alpha}, S_0))$ and $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$.

Extending the Inverse Mapping to Sets of Action Sequences. In Section 5.5, we introduced a notion of inverse mapping that maps a sequence of low-level actions back into a unique abstraction high-level action sequence it refines. We can extend this notation to apply to any set of action sequences E_l as well, i.e., $m^{-1}_{M_l}(E_l, s_l) = E_h$ if and only if $E_h = \{\vec{\alpha} \mid \vec{a} \in E_l \text{ and } M_l \models Do(m(\vec{\alpha}), s_l, do(\vec{a}, s_l))\}$.

6.1.2 The setp(P) Construct and the Monitorable Agents

In this chapter, we consider a low-level agent that only executes low-level action sequences that refine some high-level action sequences. One option is to re-use the low-level programs ANY1HLREF and ANYSEQHLREF characterized in Section 5.3. The low-level program ANY1HLREF allows the refinements of any high-level primitive action to be executed; however, observe that even if the program associated to each high-level action $m(A_i(\vec{x}))$ is situation-determined (SD), the non-deterministic branch of all of these, $|_{A_i \in \mathcal{A}_h} \pi\vec{x}.m(A_i(\vec{x}))$, may not be SD if executions of different high-level actions may share prefixes. For example, 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.

Note however that once we have finished executing at the low level a sequence $\vec{a}$ that is a refinement of some high-level action, i.e., have that $Do(\textsc{any1hlref}, s_l, do(\vec{a}, s_l))$, by Assumption 5.1, there is a unique high-level action α that the sequence $\vec{a}$ refines, i.e., such that $Do(m(\alpha), s_l, do(\vec{a}, s_l))$. This justifies using a new program construct $\mathbf{setp}(P)$ that executes a set of programs P non-deterministically without committing to which element of P is being executed unless it has to. The transition semantics of this construct is as follows:

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

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

Monitorable Agents. We assume that the low-level 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 notion:

$$\textsc{anyOne} \doteq \ |_{A_i \in \mathcal{A}_h} \ \pi \vec{x}. A_i(\vec{x}), \ \text{do any HL primitive action,}$$

$$\textsc{any} \doteq \textsc{anyOne}^*, \ \text{i.e., do any sequence of HL actions.}$$

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

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

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 $\mathcal{CR}_{M_l}(\textsc{monit}, S_0)$, where M_l is a model of the low-level BAT and $\mathcal{C}$.[32] If we

[32]Assumption 5.2 requires that any low-level situation can in fact be generated by a (partial) execution of a refinement of some high-level action sequence. Monitorable agents on the other hand, refer to an agent whose behavior is described by the MONIT program which only executes refinements of high-level actions. Thus, all executable situations reached by MONIT satisfy the

have bisimilar models, the converse follows, i.e., any executable high-level action sequence has an executable refinement at the low-level.

6.2 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 6.1.

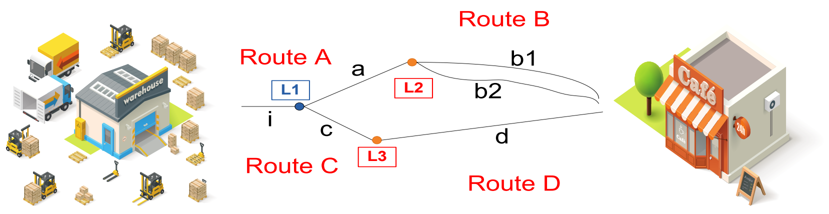

Figure 6.1: Transport Logistics Example

High-Level BAT $\mathcal{D}_h^{eg}$ At the high level, we abstract over navigation, delivery procedure details and potential causes of delay. We have actions that represent choices of major routes, delivering a shipment and possible delay. It typically takes longer for a shipment to be sent on route A followed by B compared to taking route C followed by D; on the other hand, route D may sometimes be closed for various reasons, causing a delay in the trip. Shipments may be high-priority or not, represented by a fluent $Priority(sID, s)$. Moreover, the fluent $Dest_{HL}$ specifies the destination of the shipment and the fluent At_{HL} indicates its current location.

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

$$Poss(takeRoute(sID, r, o, d), s) \equiv o \neq d \wedge At_{HL}(sID, o, s) \wedge CnRoute_{HL}(r, o, d, s)$$

$$Poss(deliver(sID), s) \equiv \exists l.Dest_{HL}(sID, l, s) \wedge At_{HL}(sID, l, s)$$

$$Poss(delay(sID), s) \equiv At_{HL}(sID, L3, s)$$

The action $takeRoute(sID, r, o, d)$ can be performed to take shipment with ID sID from origin location o to destination location d via route r (see Figure 6.1), and is executable when the shipment is initially at o and route r connects o to d; note that we refer to route X in Figure 6.1 as Rt_X. Action $deliver(sID)$ can be performed to deliver shipment sID and is executable when sID is at its destination. Both of these actions

condition of Assumption 5.2, however, there could be some situations which are not reachable by MONIT, i.e., not the result of a (partial) execution of a high-level action sequence. It is mainly a modeling choice: Assumption 5.2 constraints the basic action theory, while monitorable agents restrict the agent behavior of interest to the part of the situation tree that is monitorable.

are assumed to be controllable. Action $delay(sID)$ is an exogenous and uncontrollable action that may occur when the shipment is at location $L3$.

The high-level BAT also includes the following SSAs:

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

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

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

$$\begin{aligned}
&Priority(sID, S_0) \equiv sID = 123, \\
&Dest_{HL}(sID, l, S_0) \equiv sID = 123 \land l = Cf, \\
&At_{HL}(sID, l, S_0) \equiv sID = 123 \land l = W, \\
&A_h^u(a, S_0) \equiv \exists sID.a = delay(sID), \\
&CnRoute_{HL}(r, l_s, l_e, S_0) \equiv \\
&\qquad [(r = Rt_A \land l_s = W \land l_e = L2) \lor (r = Rt_B \land l_s = L2 \land l_e = Cf) \lor \\
&\qquad (r = Rt_C \land l_s = W \land l_e = L3) \lor (r = Rt_D \land l_s = L3 \land l_e = Cf)]
\end{aligned}$$

Low Level BAT $\mathcal{D}_l^{eg}$ At the low level, we model navigation, delivery and causes of delay in a more detailed way. The agent has a more detailed map with more locations and roads between them. 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 6.1). Performing delivery involves unloading the shipment and getting a signature. Two causes of *delay* are considered to be due to bad weather or road maintenance. The agent also takes into account that some roads can only be used at night time by trucks carrying a shipment. This condition is indicated by the fluent $NT(sID, s)$. Moreover, *Priority* shipments are refined to two types of express shipments: Express Same Day ($Exp1$) or Express 2 Days ($Exp2$).

The low-level BAT $\mathcal{D}_l^{eg}$ includes the following action precondition axioms:

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

The action $takeRoad(sID, t, o, d)$, where the agent takes shipment sID from origin location o to destination d via road t, is executable provided that t connects o to d, sID is at o; moreover, road Rd_{b2} can only be used if shipment is sent during nighttime (note that we refer to road x in Figure 6.1 as Rd_x). The action *unload* can be performed when the shipment has reached its destination; after unloading the shipment, *getSignature* can be performed. Actions *takeRoad*, *unload* and *getSignature* are assumed to be controllable actions. 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.

The low-level BAT includes the following SSAs:

$$\begin{aligned}
&Unloaded(sID, do(a, s)) \equiv a = unload(sID) \lor Unloaded(sID, s)\\
&Signed(sID, do(a, s)) \equiv a = getSignature(sID) \lor Signed(sID, s)\\
&DelayedBD(sID, do(a, s)) \equiv a = delayBD(sID) \lor DelayedBD(sID, s)\\
&DelayedRM(sID, do(a, s)) \equiv a = delayRM(sID) \lor DelayedRM(sID, s)
\end{aligned}$$

The SSA for At_{LL} is like the one for At_{HL} with *takeRoute* replaced by *takeRoad*. For the other fluents, we have SSAs specifying that they are unaffected by any actions.

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

$$\begin{aligned}
&\neg Exp1(sID, S_0),\\
&Exp2(sID, S_0) \equiv sID = 123,\\
&Dest_{LL}(sID, l, S_0) \equiv sID = 123 \land l = Cf,\\
&At_{LL}(sID, l, S_0) \equiv sID = 123 \land l = W,\\
&NT(sID, S_0) \equiv sID = 123,\\
&A_l^u(a, S_0) \equiv \exists sID(a = delayBW(sID) \lor a = delayRM(sID))
\end{aligned}$$

together with a complete specification of $CnRoad$ and $CnRoute_{LL}$.

Refinement Mapping m^{eg} We specify the relationship between the high-level and low-level BATs through the following refinement mapping m^{eg}:

$$
\begin{aligned}
&m^{eg}(takeRoute(sID, r, o, d)) = \\
&\quad (r = Rt_A \wedge CnRoute_{LL}(Rt_A, o, d))?; \\
&\quad\quad \pi t.takeRoad(sID, t, o, L1); takeRoad(sID, Rd_a, L1, d) \mid \\
&\quad (r = Rt_B \wedge CnRoute_{LL}(Rt_B, o, d))?; \\
&\quad\quad \pi t.takeRoad(sID, t, L2, d) \mid \\
&\quad (r = Rt_C \wedge CnRoute_{LL}(Rt_C, o, d))?; \\
&\quad\quad \pi t.takeRoad(sID, t, o, L1); takeRoad(sID, Rd_c, L1, d) \mid \\
&\quad (r = Rt_D \wedge CnRoute_{LL}(Rt_D, o, d))?; \\
&\quad\quad \pi t.takeRoad(sID, t, L3, d); \\
&m^{eg}(deliver(sID)) = unload(sID); getSignature(sID) \\
&m^{eg}(delay(sID)) = delayBW(sID) \mid delayRM(sID)
\end{aligned}
$$

$$
\begin{aligned}
&m^{eg}(Priority(sID)) = Exp1(sID) \vee Exp2(sID) \\
&m^{eg}(Delivered(sID)) = Unloaded(sID) \wedge Signed(sID) \\
&m^{eg}(Delayed(sID)) = DelayedBW(sID) \vee DelayedRM(sID) \\
&m^{eg}(At_{HL}(sID, loc)) = At_{LL}(sID, loc) \\
&m^{eg}(CnRoute_{HL}(r, o, d)) = CnRoute_{LL}(r, o, d) \\
&m^{eg}(Dest_{HL}(sID, l)) = Dest_{LL}(sID, l)
\end{aligned}
$$

Thus, taking route Rt_A involves first taking a road from the origin o to $L1$ and then taking another road from $L1$ to the destination d. Taking Rt_C is refined in a similar way. For other two routes, the refinement does not include taking a road to an intermediate location. The action *delay* is refined to a non-deterministic choice of either *delayBW* or *delayRM*. We refine the high-level fluent $Priority(sID)$ to the condition where either the shipment is of type $Exp1$ or $Exp2$.

6.3 Hierarchical Agent Supervision

We would like to use abstraction for supervision. We assume that supervision specification is given as a high-level program, which is quite natural.

Example 6.1 Referring back to our running example, we may have a supervision specification δ^h_{Spec} 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:

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

□

To facilitate supervision, we would like to be able to first obtain a supervisor for the high-level agent, and then use it to obtain a supervisor for the low-level agent.

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 (for proofs of our results in this chapter, see Appendix A.3):

Lemma 6.1

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

i.e., at the high level, any set of executable *atomic* actions E_S is controllable with respect to 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 α, β, and γ, all of which are executable in S_0, with the following mapping:

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

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. At the low level, $\{m(\alpha), m(\beta)\}$ is controllable but $\{m(\alpha)\}$ is not; we cannot block β without blocking α as well and vice versa. The high-level model cannot represent this kind of example by classifying individual actions as controllable or not (using $A_h^u(a_h, s)$).

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., $A_h^u(a, s)$) 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 6.1 (Local Controllability) *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$ (with respect to $M_h \sim_m M_l$) and $M_h \models \mathcal{C}$, then*

(a) for any set of ground high-level actions E_h,
$M_h \models \mathit{Controllable}(\mathbf{set}(E_h), \textsc{anyOne}, do(\vec{\alpha}, S_0))$
if and only if

there exists a set of ground low-level action sequences E_l such that
$M_l \models \mathit{Controllable}(\mathbf{set}(E_l), \textsc{oneMonit}, do(\vec{a}, S_0))$ *and* $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$*;*

(b) $M_l \models \mathit{Controllable}(\mathbf{set}(\{\epsilon\}), \textsc{monit}, do(\vec{a}, S_0))$
if and only if
$M_h \models \mathit{Controllable}(\mathbf{set}(\{\epsilon\}), \textsc{any}, do(\vec{\alpha}, S_0))$*.*

Intuitively, part (a) ensures that if we have a controllable set of atomic actions at the high level, we can always find a set of refinements of exactly these actions that is controllable 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 at the low level with respect to an agent that executes sequences of 0 or more refinements of high-level actions, 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 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 6.2 Referring back to our running example, suppose that at the low level, the agent has executed $\vec{a} = \mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_i, o, d); \mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_c, o, d)$ which corresponds to the high-level $\vec{\alpha} = \mathit{takeRoute}(\mathit{sID}, \mathit{Rt}_C, o, d)$. At the high level, the set of all executable actions is $A_h = \{\mathit{takeRoute}(\mathit{sID}, \mathit{Rt}_D, o, d), \mathit{delay}(\mathit{sID})\}$ and the only controllable subsets are A_h, $\{\mathit{delay}(\mathit{sID})\}$ and $\emptyset$. At the low level, the controllable subsets are $\{\mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_d, o, d), \mathit{delayBW}(\mathit{sID}), \mathit{delayRM}(\mathit{sID})\}$, $\{\mathit{delayBW}(\mathit{sID}), \mathit{delayRM}(\mathit{sID})\}$, 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 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 at the low level? Indeed we have been able to show that we can, when the high-level action sequences in E_h have a bounded length, as the following result shows:

Theorem 6.2 *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 (part(a) $\supset$) hold, then*

for any set of ground high-level action sequences of bounded length E_h such that $M_h \models \mathit{Controllable}(\mathbf{set}(E_h)$, ANY, $do(\vec{\alpha}, S_0)$), *there exists a set of ground low-level action sequences* E_l *such that* $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ *and*

$M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$ *and* $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$.

Note that a set of action sequences E has bounded length if there exists $K \in \mathbb{N}$ such that for all $\vec{a} \in E$, $|\vec{a}| \leq K$. Our proof is by induction on the length of the longest action sequence in E_h. We leave the unbounded case for future work.

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 6.3 *If* $M_h \sim_m M_l$ *and* $\vec{a}$ *is an m-refinement of an executable* $\vec{\alpha}$, $M_h \models \mathcal{C}$, *and Assumptions 5.1 and 6.1 (part(a)* $\subset$ *and part (b)) hold, then*

for any set of ground low-level action sequences E_l *such that* $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))$ *has bounded length,*

if $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$,

then $M_h \models Controllable(\mathbf{set}(m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))), \textsc{any}, do(\vec{\alpha}, S_0))$.

An immediate consequence of the above is that any set of high-level action sequences that is uncontrollable (with respect to (ANY in $do(\vec{\alpha}, S_0)$) has no refinement set that is controllable (with respect to MONIT in $do(\vec{a}, S_0)$), as if there was such a set, then E_h would have to be controllable by Theorem 6.3.

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 δ^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 notion precisely however, we need a way of mapping the high-level supervision specification program δ^h_{Spec}, which is SD, into a low-level program whose runs are the refinements of δ^h_{Spec}.

To support this, we extend the mapping m to a mapping m_p that maps any SD high-level program δ^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 δ^h_{Spec} to the SD low-level program that implements it $m_p(\delta^h_{Spec})$.

Example 6.3 Going back to our running example, applying m_p to the high-level specification δ^h_{Spec} given earlier yields the following low-level specification:

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

□

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

Theorem 6.4 *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold, then for any supervision specification represented by a high-level situation-determined program δ^h_{Spec},*

$$\begin{aligned}
m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_{offl}(\textsc{monit}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0)), do(\vec{a}, S_0)) = \\
\mathcal{CR}_{M_h}(mps_{offl}(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)).
\end{aligned}$$

provided that the sets of action sequences on both sides of this equation have bounded length.

Example 6.4 Referring back to our running example, the complete runs of the high-level MPS and low-level MPS are as follows:

$$\begin{aligned}
&\mathcal{CR}_{M_h}(mps_{offl}(\textsc{any}, \delta^h_{Spec}, S_0), S_0) = \\
&\qquad \{takeRoute(sID, Rt_A, o, d); takeRoute(sID, Rt_B, o, d); deliver(sID)\} \\
&\mathcal{CR}_{M_l}(mps_{offl}(\textsc{monit}, m_p(\delta^h_{Spec}), S_0), S_0) = \\
&\qquad \{[takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d); \\
&\qquad takeRoad(sID, Rd_{b1}, o, d); unload(sID); getSignature(sID)], \\
&\qquad [takeRoad(sID, Rd_i, o, d); takeRoad(sID, Rd_a, o, d); \\
&\qquad takeRoad(sID, Rd_{b2}, o, d); unload(sID); getSignature(sID)]\}
\end{aligned}$$

It is easy to confirm that the result of Theorem 6.4 holds. □

6.4 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):

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

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 $mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta_{Spec}^h), do(\vec{a}, S_0))$ obtained by mapping the supervision specification δ_{Spec}^h to the low level:

Theorem 6.5 *If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold, then for any supervision specification represented by a high-level situation-determined program δ_{Spec}^h where $\mathcal{CR}_{M_h}(\delta_{Spec}^h, do(\vec{\alpha}, S_0))$ and $m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0))$ have bounded length,*

$$\begin{aligned}
&\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) = \mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta_{Spec}^h), do(\vec{a}, S_0)), do(\vec{a}, S_0)) \\
&\textit{where } E_h^{mps} = \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)).
\end{aligned}$$

It should be clear that the hierarchically synthesized MPS $mps_i(E_h^{mps})$ will generally be much easier to compute than the low-level MPS $mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta_{Spec}^h), do(\vec{a}, S_0))$. 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 $mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0))$, but the search space for it 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 6.5 Referring back to our running example, it is clear that initially, $mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta^h_{Spec}), S_0)$ needs to consider both action sequences $[\mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_i, o, d); \mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_a, o, d)]$ and $[\mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_i, o, d); \mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_c, o, d)]$. After the latter, the uncontrollable actions $\mathit{delayRM}$ or $\mathit{delayBW}$ may happen next, which would violate the specification given that shipment 123 is of type $\mathit{Exp}2$. So it will only include the former as a prefix in the resulting MPS. $mps_i(E^{mps}_h)$ on the other hand, only needs to consider refinements of $\mathit{takeRoute}(\mathit{sID}, \mathit{Rt}_A, o, d)$, i.e., $\mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_i, o, d); \mathit{takeRoad}(\mathit{sID}, \mathit{Rd}_a, o, d)$. The high-level MPS $mps_{\mathit{offl}}(\textsc{any}, \delta^h_{Spec}, S_0)$ has already determined 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 the shipment 123 is of type $\mathit{Priority}$. $\square$

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 assumption 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 [12]. Extending our hierarchical approach to this case is left for future work.

6.5 Discussion

In this chapter, 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 [37].

Our approach is inspired by the hierarchical supervisory control of discrete event systems [172, 165, 166] (see Section 2.5.3), but the foundations of our work is different. Wonham [166] models the low-level and high-level plants as types of automata. An *aggregation* map from the low-level plant to the high-level plant is defined that summarizes strings of low-level events to a high-level event. The controllability of the high-level events are defined based on controllability of their refinements. Also, to ensure non-blocking supervision, a type of local controllability in the low-level plant is provided; moreover, the map is enhanced with a "global observer property", that ensures the strings of events in the low-level language can always be extended as their abstractions are in the high-level model. This approach does not consider low-level constraints that could be imposed on the low-level plant.

Our framework however, is based on a rich first-order logic language. In addition to actions (which abstract over programs), our high-level theory includes fluents (which abstract over formulas). Moreover, we use a *refinement* mapping that associates each high-level primitive action to a (possibly non-deterministic) ConGolog program defined over the low-level action theory that "implements" it and maps each high-level fluent to a state formula in the low-level language that "characterizes the concrete conditions" under which it holds. We use a notion of bisimulation to relate the models of the high-level and low-level theories. Our notion of local controllability is similar to that of [166]; moreover, as we show in Theorem 6.2 and Theorem 6.3, we are able to achieve the main condition with marking (MCm). Finally, through preconditions for actions, we are able to enforce local constraints on the low-level agent.

Based on the "Roman Model" approach to behavior composition in AI, Sardina and DeGiacomo [138] synthesize a controller that orchestrates the concurrent execution of library of available (non-deterministic) ConGolog programs to realize a target program not in the library. However, their controller is not maximally permissive. In related work, Yadav et al. [169], consider optimal realization of the target behavior (in the presence of uncontrollable exogenous events) when its full realization is not possible. This work does not synthesize a supervisor and uses a controller. Also, it does not assume the controllability of events to be dynamic. De Giacomo et al. [44] however, synthesize a *controller generator* that represents all possible compositions of the target behavior and may adapt reactively based on runtime feedback. In more recent work, Felli et al. [58] relate the notion of a composition controller in the "Roman Model" approach to behavior composition to that of a supervisor in SCDES. Differently from our framework which uses a rich first order language, these three approaches model behaviors/services as (non-deterministic) finite state transition systems.

In the area on norm enforcement, the approach by Alechina et al. [2] regulates multiagent systems using regimented norms. A transition system describes the behavior of a (multi-) agent system and a guard function (characterized by LTL formulae with past operators) can enable/disable options that (could) violate norms after a system history (possibly using bounded lookahead). Unlike our approach, this work does not consider uncontrollable events, and is not based on expressive first order logic language. Gabaldon [65] investigates expressing and enforcing norms in the Golog programming language. Norms are encoded as additional preconditions of actions. In work on imposing control knowledge in planning, Gabaldon [64] provides a procedure for compiling search control knowledge in nonMarkovian action theories in the situation calculus.

Control knowledge is added to precondition axioms of actions. While this approach focuses on improving the efficiency of planning, our work provides a generic framework which can have applications in many areas. Moreover, adding constraints as preconditions of actions in the latter two approaches, can not provide the flexibility and system evolvability that our framework can provide, as the supervision specification is encoded separately from agent behavior. Also, these two approaches do not consider uncontrollable actions.

Aucher [7] reformulates the results of supervisory control theory in terms of model checking problems in an epistemic temporal logic. Our work differs from this approach in that due to its first-order logic foundations, it can handle infinite object domains and infinite states. It also enables users to express the system model and the specifications in a high-level expressive language. None of the above approaches consider abstraction.

In planning, several notions of abstraction have been investigated. These include *precondition elimination abstraction*, first introduced in the context of ABSTRIPS [135]; *Hierarchical Task Networks* (HTNs) (e.g., [57]), which abstract over a set of (non-primitive) tasks; and *macro operators* (e.g., [89]), which represent meta-actions built from a sequence of action steps. Encodings of HTNs in ConGolog [63] with enhanced features like exogenous actions have also been studied. McIlraith and Fadel [112] instead, investigate planning with *complex actions* (a form of macro actions) specified as Golog programs. While these approaches focus on improving the efficiency of planning, our work provides a generic framework which can have applications in many areas. 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, they only deal with deterministic complex actions and do not provide abstraction for fluents.

Grossi and Dignum [79] use a KD45 multi modal logic corresponding to a propositional logic of contexts to model norms at different levels of abstractions. In related work, Salceda and Dignum [162] propose a method to refine abstract norms specified in the institutional regulations to concrete norms and eventually into rules and procedures represented by a PDL [114] such that the agents operating within the organization can be rewarded/punished based on existing norms. Our framework however, is based on an expressive first order logic language.

7 Abstracting Online Agent Behavior

To facilitate reasoning about agents that have complex behavior, we have developed a general framework for agent abstraction in Chapter 5. This framework focused on abstraction of agent behavior in offline executions. In this chapter, we study how we can apply this framework in the case where the agent may acquire new knowledge while executing, for example through sensing. As in Chapter 5, we assume that one has a high-level/abstract action theory, a low-level/concrete action theory, and a refinement mapping between the two. Consider for instance a travel planner agent that needs to book a seat on a certain flight. The high-level theory could abstract over details of the booking procedure, while the low-level theory could consider the detailed procedure, such as selecting flights and making payments. Similarly, the existence of upgrades from economy to business class may be abstracted into a single category of upgrade at the high level, while at the low level, different types of upgrades, such as existence of air miles rewards or promotions can be considered. The agent may not know ahead of time whether such upgrades are available; she will only learn about the existence of an upgrade after querying a web service at execution time. Therefore, we must consider agent's online executions [39, 140]. As discussed in Chapter 4, such online executions can only be defined meta-theoretically (unless one adds a knowledge operator/fluent [145]) since at every time point, the knowledge base used by the agent to deliberate about the next action may be different.

Abstraction is useful in reasoning about an agent that executes online in similar contexts as those considered when an agent executes offline. Examples of applications of abstraction of online agent behavior include hierarchical contingent planning, agent monitoring, providing high-level explanations, and online hierarchical agent supervision.

To formalize a notion of *sound abstraction in online executions*, we first identify a sufficient property for sound abstraction to persist along an online execution. We then show results extending basic properties of sound abstractions to online executions. We adapt definitions of strategies and ability to perform a task/achieve a goal to our model of online execution. Based on this, we show that under some reasonable conditions, if we have sound abstraction and the agent has a conditional plan/strategy for accomplishing a task or achieving a goal at the high level, then we can refine it into a low-level strategy piecewise, and the resulting low-level strategy is guaranteed to achieve the refinement of the goal.

7.1 A Travel Planning Example

For our running example, we consider a travel planner agent, that can book a seat on a certain flight for a customer with client ID "$C1$". We assume the agent communicates with a single web service to gather the required information and make bookings. We further assume that the agent can always book an economy seat (Ec) on a flight; however, to book a business seat (Bz), the customer must be eligible for some kind of upgrade from economy class. The agent initially does not know whether the customer is eligible for an upgrade. For simplicity, we ignore the dates of the trip and assume seats are always available at the beginning of the process.

The high-level BAT $\mathcal{D}_h^{eg}$. At the high level, we abstract over details of the booking procedure. The agent first queries the web service to learn whether an upgrade is available for a customer. This is followed by a reply from the web service, indicating whether or not the upgrade is available, after which, the agent can make a booking.

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

$$\begin{aligned}
&Poss(qryUpg(c), s) \equiv \neg QrdUpg(c, s) \wedge \neg\exists c'.(QrdUpg(c') \wedge \neg RcvdRep(c'))\\
&Poss(repUpg(c, x), s) \equiv \neg RcvdRep(c, s) \wedge QrdUpg(c, s) \wedge\\
&\qquad\qquad\qquad\qquad (Upg(c, s) \wedge x = 1 \vee \neg Upg(c, s) \wedge x = 0)\\
&Poss(book(c, cls, f), s) \equiv \neg Booked(c, cls, f, s) \wedge ((cls = Bz) \supset Upg(c, s)) \wedge\\
&\qquad\qquad\qquad\qquad \neg\exists c'.(QrdUpg(c') \wedge \neg RcvdRep(c'))
\end{aligned}$$

The fluent $Upg(c, s)$ holds when an upgrade for customer c is possible. $qryUpg(c)$ is an ordinary action that is used to query if $Upg(c, s)$ holds, and is executable for any customer c, unless the web service has already been queried whether $Upg(c, s)$ holds for that customer (indicated by $\neg QrdUpg(c, s)$ in the precondition). Also, it should not be the case that the existence of an upgrade for a customer c' has been queried, but no reply from the web service has been received (indicated by the fluent $RcvdRep(c')$). $repUpg(c, x)$ is an exogenous action that informs the agent whether $Upg(c, s)$ holds through its precondition axiom: $repUpg(c, 1)$ is executed if $Upg(c, s)$ holds, and otherwise, $repUpg(c, 0)$ is executed. After the action $repUpg(c, x)$ has been executed, the fluent $RcvdRep(c, s)$ is set to indicate that the reply from web service has been received. Thus, $\neg RcvdRep(c, s)$ in the preconditions specifies that no reply has been received yet. The fluent $QrdUpg(c, s)$ in the precondition indicates that action $qryUpg(c)$ has already been performed. $book(c, cls, f)$ is an ordinary action that can be performed to book a business or economy class seat (indicated by the cls argument) on a flight f for customer with client id c and is executable if the ticket has not been booked already; moreover, for booking Bz seats, an upgrade must be available. Also, it should not be the case that the existence of an upgrade for a customer c' has been queried, but no reply from the web service has been received.

The high-level BAT also includes the following successor state axioms:

$$\begin{aligned}
&QrdUpg(c, do(a, s)) \equiv a = qryUpg(c) \vee QrdUpg(c, s)\\
&RcvdRep(c, do(a, s)) \equiv a = repUpg(c, x) \vee RcvdRep(c, s)\\
&Booked(c, cls, f, do(a, s)) \equiv a = book(c, cls, f) \vee Booked(c, cls, f, s)
\end{aligned}$$

$QrdUpg$ for customer c holds in a situation resulting from performing action a in situation s if and only if the action a was $qryUpg(c)$, or $QrdUpg$ was already true in situation s. The successor state axioms for $RcvdRep$ and $Booked$ are defined similarly. For Upg, we have a successor state axiom specifying that it is unaffected by any action.

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

$$\begin{aligned}
&\neg QrdUpg(C1, S_0),\\
&\neg RcvdRep(C1, S_0),\\
&\forall cls, f.\neg Booked(C1, cls, f, S_0)
\end{aligned}$$

Note that in the initial state, it is not known whether $Upg(C1, S_0)$ holds.

The low-level BAT $\mathcal{D}_l^{eg}$. At the concrete level, we consider the process of booking a ticket in more details. Here, two types of upgrades exit: using air miles rewards and using a promotion. Hence, querying and getting replies about the availability of an upgrade involve more specific querying and receiving replies about existence of promotions as well as air miles rewards availability. Moreover, performing a booking involves first selecting a flight and then making a payment.

In formalizing the low-level theory, we want to ensure that all low-level executions are refinements of high-level action sequences, we need to assume a number of preconditions for each action. For instance, $qryUpg(c)$ is refined by $qryAM(c); qryPr(c)$, thus $qryPr(c)$ must only be executable when $qryAM(c)$ has just been done and no other actions can be allowed in between those. Similarly for the sequences $repAM(c, x); repPr(c, x)$ refining $repUpg(c, x)$ and $selectFlt(c, cls, f); pay(c, cls, f)$ refining $book(c, cls, f)$ (we will define the refinement mapping m^{eg} later in this section).

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

$$\begin{aligned}
&Poss(qryAM(c), s) \equiv \neg QrdAM(c, s) \wedge \\
&\qquad \neg\exists c', f', cls'.(Querying(c') \vee Replying(c') \vee Booking(c', cls', f')) \wedge \\
&\qquad \neg\exists c'.(QrdAM(c') \wedge QrdPr(c') \wedge \neg(RcvdRepAM(c') \wedge RcvdRepPr(c'))) \\
&Poss(qryPr(c), s) \equiv \neg QrdPr(c, s) \wedge QrdAM(c, s) \wedge \\
&\qquad \neg\exists c'.(QrdAM(c') \wedge QrdPr(c') \wedge \neg(RcvdRepAM(c') \wedge RcvdRepPr(c'))) \\
&Poss(repAM(c, x), s) \equiv \neg RcvdRepAM(c, s) \wedge QrdAM(c, s) \wedge QrdPr(c, s) \wedge \\
&\qquad (AirMiles(c, s) \wedge x = 1 \vee \neg AirMiles(c, s) \wedge x = 0) \wedge \\
&\qquad \neg\exists c', f', cls'.(Querying(c') \vee Replying(c') \vee Booking(c', cls', f')) \\
&Poss(repPr(c, x), s) \equiv \neg RcvdRepPr(c, s) \wedge RcvdRepAM(c, s) \wedge QrdAM(c, s) \wedge QrdPr(c, s) \wedge \\
&\qquad (Promotion(c, s) \wedge x = 1 \vee \neg Promotion(c, s) \wedge x = 0) \\
&Poss(selectFlt(c, cls, f), s) \equiv \neg Selected(c, cls, f, s) \wedge \\
&\qquad ((cls = Bz) \supset AirMiles(c, s) \vee Promotion(c, s)) \wedge \\
&\qquad \neg\exists c', f', cls'.(Querying(c') \vee Replying(c') \vee Booking(c', cls', f')) \wedge \\
&\qquad \neg\exists c'.(QrdAM(c') \wedge QrdPr(c') \wedge \neg(RcvdRepAM(c') \wedge RcvdRepPr(c'))) \\
&Poss(pay(c, cls, f), s) \equiv \neg Paid(c, cls, f, s) \wedge Selected(c, cls, f, s) \wedge \\
&\qquad \neg\exists c'.(QrdAM(c') \wedge QrdPr(c') \wedge \neg(RcvdRepAM(c') \wedge RcvdRepPr(c')))
\end{aligned}$$

$AirMiles(c, s)$ and $Promotion(c, s)$ are fluents that indicate respectively whether using air miles rewards and using a promotion are possible for customer c. $qryAM(c)$ and $qryPr(c)$ are ordinary actions that are used to query whether $AirMiles(c, s)$ and $Promotion(c, s)$ hold respectively. $qryAM(c)$ is executable for any customer c, unless it has already been queried whether $AirMiles(c, s)$ holds for that customer (indicated by $\neg QrdAM(c, s)$ in the precondition). Moreover, the agent can only perform $qryAM(c)$, if the process of querying a web service about upgrades, receiving a reply or booking a ticket for any customer c', indicated by abbreviations $Querying(c') \doteq QrdAM(c') \wedge \neg QrdPr(c')$, $Replying(c') \doteq RcvdRepAM(c') \wedge \neg RcvdRepPr(c')$ and $Booking(c', cls, f) \doteq Selected(c', cls, f) \wedge \neg Paid(c', cls, f)$ respectively, have not been already initiated (we provide an overview of these fluents below). Moreover, it should not be the case that the existence of air miles rewards and promotions have already been queried for a customer, but the replies from the web service are still pending. The action $qryPr(c)$ is executable for a customer c if it has not already been queried whether a promotion exists for c (i.e., $Promotion(c, s)$ holds) and $qryAM(c)$ has already been performed for c. Similar to above, there should not be any customer c' for whom the existence of air miles rewards or promotions have been queried, and no replies from the web service have been received. $repAM(c, x)$ (resp. $repPr(c, x)$) is an exogenous action that informs the agent whether $AirMiles(c, s)$ (resp. $Promotion(c, s)$) holds through its precondition axiom: $repAM(c, 1)$ is executed if $AirMiles(c, s)$ holds, and otherwise, $repAM(c, 0)$ is executed (and similarly for $repPr(c, x)$). After the action $repAM(c, x)$

(resp. $repPr(c, x)$) has been executed, the fluent $RcvdRepAM(c, s)$ (resp. $RcvdRepPr(c, s)$) is set to indicate that the reply from web service regarding existence of air miles (resp. promotion) has been received. Thus, $\neg RcvdRepAM(c, s)$ (resp. $\neg RcvdRepPr(c, s)$) in the precondition indicates that a reply has not been received yet. The fluents $QrdAM(c, s)$ and $QrdPr(c, s)$ in the precondition of $repAM(c, x)$ and $repPr(c, x)$ indicate that the actions $qryAM(c)$ and $qryPr(c)$ have already been performed. Moreover, the action $repAM(c, x)$ can only be executed if the process of querying a web service about upgrades, receiving a reply or booking a ticket for any customer has not been already initiated. $selectFlt(c, cls, f)$ is an ordinary action that selects a seat on a flight f and is executable when a seat on that flight has not been selected for that customer before; and, $RcvdRepPr(c, s)$ has been received (which also indicates that $RcvdRepAM(c, s)$ has been received too, since it is the precondition of $qryPr(c)$). Moreover, for booking Bz seats, either the customer should have sufficient air miles or a promotion exists. Also, the agent can perform $selectFlt(c, cls, f)$ if the process of querying a web service about upgrades, receiving a reply or booking a ticket for any customer has not been already initiated. Finally, it should not be the case that for any customer c' the web services has been queried about air miles rewards and promotions, but no replies from the web service have been received yet. After the seat/flight is selected, payments can be made by using the ordinary action *pay*, unless the fees have already been paid (specified by $\neg Paid(c, cls, f, s)$ in the precondition).

The low-level BAT also includes the following successor state axioms:

$$\begin{aligned}
&QrdAM(c, do(a, s)) \equiv a = qryAM(c) \vee QrdAM(c, s)\\
&QrdPr(c, do(a, s)) \equiv a = qryPr(c) \vee QrdPr(c, s)\\
&RcvdRepAM(c, do(a, s)) \equiv a = repAM(c) \vee RcvdRepAM(c, s)\\
&RcvdRepPr(c, do(a, s)) \equiv a = repPr(c) \vee RcvdRepPr(c, s)\\
&Selected(c, cls, f, do(a, s)) \equiv a = selectFlt(c, cls, f) \vee Selected(c, cls, f, s)\\
&Paid(c, cls, f, do(a, s)) \equiv a = pay(c, cls, f) \vee Paid(c, cls, f, s)
\end{aligned}$$

For fluents $AirMiles(c, s)$ and $Promotion(c, s)$, we have successor state axioms specifying that they are unaffected by any action.

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

$$\begin{aligned}
&\neg QrdAM(C1, S_0),\\
&\neg QrdPr(C1, S_0),\\
&\neg RcvdRepAM(C1, S_0),\\
&\neg RcvdRepPr(C1, S_0),\\
&\forall cls, f. \neg Selected(C1, cls, f, S_0),\\
&\forall cls, f. \neg Paid(C1, cls, f, S_0)
\end{aligned}$$

Note that it is not known whether $Promotion(C1, S_0)$ or $AirMiles(C1, S_0)$ hold.

Refinement Mapping m^{eg}. We specify the relationship between the high-level and low-level BATs through the following refinement mapping m^{eg}:

$$
\begin{aligned}
&m^{eg}(qryUpg(c)) = qryAM(c); qryPr(c)\\
&m^{eg}(repUpg(c,x)) =\\
&\quad (x=0)?;\\
&\qquad\quad (repAM(c,0); repPr(c,0))\\
&\quad (x=1)?;\\
&\qquad\quad (repAM(c,0); repPr(c,1) \mid\\
&\qquad\quad repAM(c,1); repPr(c,0) \mid\\
&\qquad\quad repAM(c,1); repPr(c,1))\\
&m^{eg}(book(c,cls,f)) = selectFlt(c,cls,f); pay(c,cls,f)\\
&m^{eg}(Upg(c)) = AirMiles(c) \lor Promotion(c)\\
&m^{eg}(QrdUpg(c)) = QrdAM(c) \land QrdPr(c)\\
&m^{eg}(RcvdRep(c)) = RcvdRepAM(c) \land RcvdRepPr(c)\\
&m^{eg}(Booked(c,cls,f)) = Selected(c,cls,f) \land Paid(c,cls,f)
\end{aligned}
$$

Thus $qryUpg(c)$ is refined to first performing $qryAM(c)$ followed by $qryPr(c)$.[33] Performing action $repUpg(c, 0)$ involves sequence of actions $repAM(c,0); repPr(c,0)$, while performing $repUpg(c,1)$ is refined as a non-deterministic choice of sequences of actions $repAM(c,0); repPr(c,1)$ or $repAM(c,1); repPr(c,0)$ or $repAM(c,\ 1); repPr(c,1)$. $Upg(c)$ is mapped into the availability of upgrade through $AirMiles(c)$ or a $Promotion(c)$. The refinements of other fluents and actions are straightforward.

By using Theorem 5.9, we can confirm that for the initial theories, $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_l^{eg}$ relative to refinement mapping m^{eg}. $\mathcal{D}_l^{S_0}$ entails the "translation" of all the facts about the high-level fluents $QrdUpg$, $RcvdRep$ and $Booked$ that are in $\mathcal{D}_h^{S_0}$. Moreover, $\mathcal{D}_l^{eg}$ entails that the mapping of the preconditions of the high-level actions $qryUpg$, $repUpg$, and $book$ correctly capture the executability conditions of their refinements. $\mathcal{D}_l^{eg}$ also entails the mapped high-level successor state axiom for fluent $QrdUpg$ and action $qryUpg$, for fluent $RcvdRep$ and action $repUpg$, and for fluent $Booked$ and action $book$; the other high-level actions don't affect these fluents. Upg has a successor state axiom that is not affected by any actions. Thus, $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_l^{eg}$ relative to m^{eg}.

In the above example the programs refining high-level actions are quite simple. But note that refinements of high-level actions may contain sensing actions that acquire knowledge that remains "local" to the low-level

[33]Note that in the refinement of $qryUpg(c)$, the $qryAM(c)$ and $qryPr(c)$ actions could be done concurrently; we prescribe a specific sequencing to keep the example simple.

theory, such that the high-level agent does not need to know about them. For instance, one could extend the example so that in the concrete theory the action *book* is refined as: $m(\mathit{book}(c, cls, f)) = \mathit{selectFlt}(c, cls, f)$; $[CC(c, s)?; payCC(c, cls, f) \mid \neg CC(c, s)?; payDB(c, cls, f)]$, where $payCC$ and $payDB$ are ordinary actions that specify payments for the ticket fees by credit and debit card respectively. The fluent $CC(c, s)$ indicates customer's preference for paying by credit card. Thus, informally, the action *book* is refined in the concrete theory by first selecting the flight, and then, based on the customer's preference, paying by credit card or debit card. As in the initial theory it may not be known whether $CC(c, s)$ holds for customer c, the low-level agent can use sensing to learn this information. Sensing for this fluent can be defined similarly to other fluents such as $AirMiles(c, s)$. The knowledge acquired however, remains local to the low level, and is not represented by any high-level fluents.

7.2 Sound Abstraction in Online Executions

How can we use abstraction in agents that execute online and acquire new information during a run? The agent's knowledge base/theory is updated when it acquires new information. The first question is whether a sound abstraction remains so when that happens.

First, note that the updated theory $\mathcal{D} \cup \{\mathit{Executable}(\vec{a}, S_0)\}$ after the agent has executed the ground action sequence $\vec{a}$ is not strictly speaking a BAT. However, it is easy to show that $\mathit{Executable}(\vec{a}, S_0)$ is equivalent given the BAT to a regressable formula $\psi_{\vec{a}}$, which can be obtained by expanding the definition of *Executable* and replacing all the *Poss* atoms in the result by the right-hand side of the relevant action precondition axiom. If we obtain the regression of this formula, i.e., $\mathcal{R}(\psi_{\vec{a}})$, then $\mathcal{D} \cup \{\mathcal{R}(\psi_{\vec{a}})\}$ is a BAT. We have that $\mathcal{D} \models \mathit{Executable}(\vec{a}, S_0) \equiv \psi_{\vec{a}}$ and thus by the regression theorem (Theorem 4.5.5 in [132]) $\mathcal{D} \models \mathit{Executable}(\vec{a}, S_0) \equiv \mathcal{R}(\psi_{\vec{a}})$. So we can apply the notion of sound abstraction to such updated theories by assuming that we regress the update formula, which we will do from now on.

Now let's get back to our question of whether a sound abstraction remains so after an action is executed online and the theory is updated. Consider the following example.

Example 7.1 Suppose that we have a high-level exogenous action a_h which is executable if and only if P_h holds. The refinements of P_h and a_h are defined as $m(P_h) = P_l$ and $m(a_h) = P_l?; a_l$ respectively, where a_l is a low-level exogenous action. Moreover, assume that at the low level, action a_l is always executable, i.e., $Poss(a_l) \equiv \texttt{True}$. Initially, it is unknown whether P_h holds at the high level and similarly for P_l at the low level. It is easy to check that the high-level theory $\mathcal{D}_h$ is a sound abstraction of the low-level theory $\mathcal{D}_l$ with respect to the mapping m. However, if a_h and its refinement a_l occur and the theories are updated, then $\mathcal{D}_h \cup \{Poss(a_h, S_0)\}$ is no longer a sound abstraction of $\mathcal{D}_l \cup \{Poss(a_l, S_0)\}$ with respect to m. In situation $do(a_h, S_0)$, the high-level agent has learned that P_h holds, as $Poss(a_h, S_0)$ has been added to the theory.

However, at the low level, adding $Poss(a_l, S_0)$ has no effect and in situation $do(a_l, S_0)$, it is still unknown whether P_l holds. The updated low-level theory has a model where $\neg P_l$ holds, which has no bisimilar model in the updated high-level theory. $\square$

Thus a sound abstraction does not always remain so when we update the theories after an action is executed online. For the sound abstraction to persist, we need to ensure that the low-level theory acquires as much information as the high-level theory. For the above example, we must ensure that the low level learns that $m(P_h)$, i.e., P_l, holds. When a high-level action occurs, the high level learns that this action was executable. If the low level also learns that it has in fact just executed a refinement of this high-level action, then we can be certain that it has acquired as much information as the high level. We can generalize this condition to action sequences and prove that it is a sufficient condition for the sound abstraction to persist (for proofs of our results in this chapter, see Appendix A.4):

Theorem 7.1 *(Persistence of Sound Abstractions) If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to refinement mapping m and $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$ holds, then $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to m.*

In the above, the condition $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$ ensures that after execution of a sequence of actions $\vec{\alpha}$ by the high-level agent/BAT and its refinement $\vec{a}$ by the low-level agent/BAT, the low level knows that it has just executed a refinement of $\vec{\alpha}$ and has learned as much information as the high level, and thus, we still have a sound abstraction.

The sufficient condition identified above to guarantee that we continue to have a sound abstraction, i.e., essentially that the low-level agent is aware of the high-level actions it executes, does not seem overly difficult to satisfy in practice. For instance, it is easy to modify the program associated to the high-level action a_h to satisfy the condition: one can define $m(a_h) = P_l?; confirm_{P_l}; a_l$, where $confirm_{P_l}$ is a new exogenous action that has P_l as precondition and no effects; this new action ensures that the agent executing $m(a_h)$ learns that $m(P_h)$, i.e., P_l, holds. Generally, it seems that we must ensure that whenever a test succeeds in an execution of a refinement of a high-level action, the low-level agent knows (i.e., the updated theory entails) afterwards that the test condition holds. One way to satisfy the condition would be to map high-level actions into "self-sufficient programs" in the sense of [113].

Example 7.2 We can show that along all online executions of refinements of high-level actions from the running example the sufficient condition holds, and the updated high-level theory remains a sound abstraction of the updated low-level theory. That is, for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon \rangle \rightarrow^*_{\vec{\alpha}} \langle \delta_h, \vec{\alpha} \rangle$ and $\langle \delta_h, \vec{\alpha} \rangle^{\checkmark}$ for some δ_h and $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l, our running example satisfies the condition $\mathcal{D}^{eg}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models$

$Do(m(\vec{\alpha}), S_0,\ do(S_0, \vec{a}))$ and thus the high-level theory $\mathcal{D}_h^{eg}$ remains a sound abstraction of the low-level theory $\mathcal{D}_l^{eg}$ relative to the mapping m^{eg} as the agent acquires new knowledge. To show this, it is sufficient to show that for all ground low-level action sequences $\vec{a}$ and $\vec{b}$, and all high-level actions β such that $\mathcal{D}_l^{eg} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge Do(m(\beta), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))\}$ is satisfiable, we have that $\mathcal{D}_l^{eg} \cup \mathcal{C} \cup \{Executable(do(\vec{a}\vec{b}, S_0))\} \models Do(m(\beta), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$.

For the high-level action $book(c, cls, f)$, this condition trivially holds since it is refined as the sequence of low-level actions $selectFlt(c, cls, f); pay(c, cls, f)$. The same holds for high-level action $qryUpg(c)$, as it is mapped to the sequence of actions $qryAM(c); qryPr(c)$. For the high-level action $repUpg(c, x)$, we have that x must be either 0 or 1 by the precondition axiom. When $x = 0$, there is one possible refinement which is $repAM(C1, 0); repPr(C1, 0)$, otherwise, there are three possible refinements: $repAM(C1, 1); repPr(C1, 0)$, $repAM(C1, 0); repPr(C1, 1)$ and $repAM(C1, 1); repPr(C1, 1)$. In each case, the low-level agent knows she has performed refinement of the associated high-level action.

$\square$

We can use our condition for the persistence of sound abstraction to extend the results of Chapter 5 on sound abstractions to use the knowledge acquired in an online execution. We can show that if the high-level agent executes action sequence $\vec{\alpha}$ online and the low-level agent is aware that $\vec{a}$ is a refinement of $\vec{\alpha}$, and if the high level agent then knows it can execute another action sequence $\vec{\beta}$ to achieve ϕ, then the low-level agent also knows after executing $\vec{a}$ online that there exists a refinement $\vec{b}$ of $\vec{\beta}$ that achieves $m(\phi)$:

Theorem 7.2 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, $\mathcal{D}_l \cup C \cup \{Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))\}$ is satisfiable, and $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$ for some ground high-level action sequence $\vec{\alpha}$ and ground low-level action sequence $\vec{a}$. Then we have that for any ground high-level action sequence $\vec{\beta}$ and high-level situation-suppressed formula ϕ, if*

$$\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \phi[do(\vec{\alpha}\vec{\beta}, S_0)],$$

then

$$\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \exists s. Do(m(\vec{\beta}), do(\vec{a}, S_0), s) \wedge m(\phi)[s].$$

A special case of the above is when $\vec{\beta}$ is the empty action sequence. Then under the theorem's conditions, we have that if the high-level agent knows that ϕ holds after executing $\vec{\alpha}$ online, then the low-level agent knows that $m(\phi)$ holds after executing refinement $\vec{a}$ of $\vec{\alpha}$ online.

Example 7.3 Returning to our running example, suppose the action sequence $\vec{\alpha} = [qryUpg(C1), repUpg(C1, 1)]$ has been executed at the high level, and that action sequence $\vec{a} = [qryAM(C1), qryPr(C1), repAM(C1, 1),$

$repPr(C1, 0)]$ has been performed at the low level. We have that $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_l^{eg}$ relative to mapping m^{eg}, and $\mathcal{D}_l^{eg} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$. Thus, since at the high level $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Upg(C1, do(\vec{\alpha}, S_0))$ and that it is known that action $book(C1, Bz, f)$ is possible for any flight f, at the low level we have that $\mathcal{D}_l^{eg} \cup C \cup \{Executable(do(\vec{a}, S_0))\} \models AirMiles(C1, do(\vec{a}, S_0)) \vee Promotion(C1, do(\vec{a}, S_0))$ (i.e., $m(Upg(C1)[do(\vec{a}, S_0)])$ and it is also known that the sequence of actions $\vec{b} = selectFlt(C1, Bz, f); pay(C1, Bz, f)$ is executable for f. Furthermore, since after executing the action $book(C1, Bz, f)$ at the high level the goal $Booked(C1, Bz, f)$ is achieved, at the low level, after performing the action sequence $\vec{b}$, the refinement of the goal $Booked(C1, Bz, f)$ is achieved (i.e., $Selected(C1, Bz, f) \wedge Paid(C1, Bz, f)$). $\square$

Another important property of sound abstractions is that if the low-level agent/BAT thinks that a refinement of $\vec{\alpha}$ may occur (with $m(\phi)$ holding afterwards), the high-level agent/BAT also thinks that $\vec{\alpha}$ may occur (with ϕ holding afterwards); if such a refinement actually occurs it will thus be consistent with the high-level theory:

Proposition 7.3 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, $\mathcal{D}_l \cup C \cup \{Executable(do(\vec{a}, S_0))\}$ is satisfiable, and $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$ for some ground high-level action sequence $\vec{\alpha}$ and ground low-level action sequence $\vec{a}$. Then we have that for any ground high-level action sequence $\vec{\beta}$ and any high-level situation-suppressed formula ϕ, if $\mathcal{D}_l \cup C \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s. Do(m(\vec{\beta}), do(\vec{a}, S_0), s) \wedge m(\phi)[s]\}$ is satisfiable, then $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \phi[do(\vec{\alpha}\vec{\beta}, S_0)]\}$ is also satisfiable.*

The above proposition follows immediately from Corollary 5.3.

Example 7.4 Referring back to our running example, suppose the sequence of actions $\vec{a} = [qryAM(C1), qryPr(C1)]$ has occurred at the low level, and the action sequence $\vec{\alpha} = qryUpg(C1)$ has been executed at the high level. We have that $\mathcal{D}_h^{eg}$ is a sound abstraction of $\mathcal{D}_l^{eg}$ relative to mapping m^{eg} and $\mathcal{D}_l^{eg} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$. Thus since at the low level both $\mathcal{D}_l^{eg} \cup C \cup \{Executable(do(\vec{a}, S_0)) \wedge (AirMiles(C1, do(\vec{a}, S_0)) \vee Promotion(C1, do(\vec{a}, S_0)))\}$ and $\mathcal{D}_l^{eg} \cup C \cup \{Executable(do(\vec{a}, S_0)) \wedge \neg(AirMiles(C1, do(\vec{a}, S_0)) \vee Promotion(C1, do(\vec{a}, S_0)))\}$ are satisfiable, then at the high level, both $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge (Upg(C1, do(\vec{\alpha}, S_0))\}$ and $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge \neg(Upg(C1, do(\vec{\alpha}, S_0))\}$ are satisfiable. Similarly, since both $\mathcal{D}_l^{eg} \cup C \cup \{Executable(do(\vec{a}\vec{b_1}, S_0))\}$ where $\vec{b_1} = [repAM(C1, 1); repPr(C1, 0)]$ and $\mathcal{D}_l^{eg} \cup C \cup \{Executable(do(\vec{a}\vec{b_2}, S_0))\}$ where $\vec{b_2} = [repAM(C1, 0); repPr(C1, 0)]$ are satisfiable at the low level, both $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(repUpg(C1, 1), do(\vec{\alpha}, S_0))\}$ and $\mathcal{D}_h^{eg} \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(repUpg(C1, 0), do(\vec{\alpha}, S_0))\}$ are satisfiable at the high level.

$\square$

Observe that online executability works differently with respect to abstraction compared to offline executability, making it more difficult to reason about the executability of a sequence of actions at one level compared to the other. For instance, while an execution of a refinement of a high-level action may be possible in the concrete theory, the execution of the high-level action may not be possible in the corresponding situation in the abstract theory. Consider the following example:

Example 7.5 Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Assume that at the high level, we have an ordinary action α that is executable when P_h holds, i.e., $Poss(\alpha, s) \equiv P_h(s)$, and that at the high level, both $D_h \cup \{P_h(S_0)\}$ and $D_h \cup \{\neg P_h(S_0)\}$ are satisfiable. Thus we have that $D_h \not\models Poss(\alpha, S_0)$. Suppose that we have the mapping: $m(P_h) = P_l$ and $m(\alpha) = a$, and that at the low level, the ordinary action a is executable when P_l holds, i.e., $Poss(a, s) \equiv P_l(s)$, and furthermore it is known that $P_l(S_0)$ holds. Therefore, $\mathcal{D}_l \models Poss(a, S_0)$ and $\mathcal{D}_l \models \exists s.Do(m(\alpha), S_0, s)$. Thus while at the low level executing action a is possible, at the high level there is no online transition that involves α. □

Moreover, while it may be possible to execute a high-level action online in the abstract theory, there may be no online execution of any of its refinements in the corresponding situation in the concrete theory, as the following example shows:

Example 7.6 Suppose that D_h is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Assume that at the high level, we have an exogenous action α that is executable when P_h holds, i.e., $Poss(\alpha, s) \equiv P_h(s)$, and that at the high level, both $D_h \cup \{P_h(S_0)\}$ and $D_h \cup \{\neg P_h(S_0)\}$ are satisfiable. At the high level, we have an online execution of α since $D_h \cup Poss(\alpha, S_0)$ is satisfiable. Suppose that we have the following mapping: $m(P_h) = P_l$, and $m(\alpha) = a$, and that at the low level, a is an exogenous action that is executable when P_l holds, i.e., $Poss(a, s) \equiv P_l(s)$, and furthermore it is known that $\neg P_l$ holds. Thus while at the high level an online execution involving α exists, at the low level there is no online execution of a i.e., $m(\alpha)$ since $\mathcal{D}_l \models \neg Poss(a, S_0)$ and $\mathcal{D}_l \models \neg\exists s.Do(m(\alpha), S_0, s)$. □

Another issue is that even if there is an online execution at the high level with a refinement that is online executable at the low level which achieves a goal or performs a task, if such an execution contains exogenous actions, the agent has no control over which exogenous actions are performed. Thus the agent may not be able to ensure that the goal is achieved or the task is performed. Consider the following example:

Example 7.7 In our running example, at the high level, we have the online execution of action sequence $\vec{\alpha} = qryUpg(C1); repUpg(C1, 1); book(C1, Bz, F1)$ that achieves the goal $Booked(C1, Bz, F1)$ in the initial configuration, i.e., booking a business seat for customer $C1$ on flight $F1$. At the low level, there is an online execution of action sequence $\vec{a} = qryAM(C1); qryPr(C1); repAM(C1, 1); repPr(C1, 0); selectFlt(C1, Bz, F1);$

$pay(C1, Bz, F1)$ that is a refinement of $\vec{\alpha}$ and achieves the refinement of the goal, i.e., $Selected(C1, Bz, F1) \wedge Paid(C1, Bz, F1)$. Since in both of these executions, there are configurations where other exogenous actions are possible (e.g., $repUpg(C1, 0)$ at $do(qryUpg(C1), S_0)$) the agent has no way of ensuring that the environment will execute an action that leads to the goal; for instance, if the environment chooses to execute $repUpg(C1, 0)$ at $do(qryUpg(C1), S_0)$, i.e., that an upgrade is not available, there is no way to ensure a business ticket is booked for customer $C1$. □

Thus, the fact that some online execution at the high level has a refinement at the low level that is executable and achieves a goal is not sufficient for the agent to be *able* to ensure a goal. She needs to have a *strategy* that ensures achieving the goal no matter how the environment behaves. To address this issue, in the next section we develop an account of strategies, agent ability and contingent planning.

7.3 Contingent Planning

Ability refers to the agent having or being able to acquire the necessary knowledge to achieve a goal or perform a task. An agent is able to perform a task/achieve a goal if she can always choose an action that leads to successful completion of the task/achievement of the goal no matter how the environment behaves. To be *able to* perform a task/achieve a goal, the agent needs to have a *strategy* that she can follow to successfully complete the task/achieve the goal, where the strategy specifies how the agent should continue to act after the environment performs some action in response to what has occurred so far. Note that we assume environment actions are fully observable. Ability is similar to the concept of *conditional* or *contingent planning* [121, 66], where agents operating online in dynamic and incompletely known environments need to construct plans/strategies that prescribe different behaviors depending on new information acquired (e.g., as a result of sensing) to ensure they achieve their goals/execute their tasks.

In formalizing ability and strategies, we need a number of assumptions. The first assumption is that it is known that in any situation that can be reached, either all the executable actions are exogenous or all the executable actions are agent's actions, i.e., the agent and the environment act only when it is their turn:

Assumption 7.1 (Turn Taking) *For $\mathcal{D} \in \{\mathcal{D}_h, \mathcal{D}_l\}$, we have that*

$$\mathcal{D} \models \forall s. \neg[(\bigvee_{A \in \mathcal{A}^e} \exists \vec{x}. Poss(A(\vec{x}), s)) \wedge (\bigvee_{A \in \mathcal{A}^o} \exists \vec{x}. Poss(A(\vec{x}), s))]$$

where $\mathcal{A}^o$ (resp. $\mathcal{A}^e$) represents the ordinary (resp. exogenous) set of action types.

Note that if at some point in the process, the environment (resp. agent) may or may not perform an action, then we can model the case where it does not by having it execute a "no-op" exogenous (resp.

ordinary) action, whose only effect is to advance the program to a configuration where it is the agent's (resp. environment's) turn to act.

Example 7.8 Going back to our running example, to clearly indicate when it is agent's turn to act and when it is the environment's, we can define the following condition on situations at the abstract level: $EnvTurn_{HL}(s) \doteq \exists c.(QrdUpg(c) \wedge \neg RcvdRep(c))[s]$. It is easy to show that ordinary actions can only be performed when this condition is false, and exogenous action can only be performed when this condition is true:

$$\begin{aligned}
&\mathcal{D}_h^{eg} \models Poss(qryUpg(c), s) \supset \neg EnvTurn_{HL}(s)\\
&\mathcal{D}_h^{eg} \models Poss(book(c, cls, f), s) \supset \neg EnvTurn_{HL}(s)\\
&\mathcal{D}_h^{eg} \models Poss(repUpg(c, x), s) \supset EnvTurn_{HL}(s)
\end{aligned}$$

We can define a similar condition on low-level situations: $EnvTurn_{LL}(s) \doteq \exists c.(QrdAM(c) \wedge QrdPr(c)) \wedge \neg(RcvdRepAM(c) \wedge RcvdRepPr(c))[s]$. Again we can show that ordinary actions are only executable when this condition is false, and exogenous actions are executable only if this condition is true:

$$\begin{aligned}
&\mathcal{D}_l^{eg} \models Poss(qryAM(c), s) \supset \neg EnvTurn_{LL}(s)\\
&\mathcal{D}_l^{eg} \models Poss(qryPr(c), s) \supset \neg EnvTurn_{LL}(s)\\
&\mathcal{D}_l^{eg} \models Poss(select(c, cls, f), s) \supset \neg EnvTurn_{LL}(s)\\
&\mathcal{D}_l^{eg} \models Poss(pay(c, cls, f), s) \supset \neg EnvTurn_{LL}(s)\\
&\mathcal{D}_l^{eg} \models Poss(repAM(c, x), s) \supset EnvTurn_{LL}(s)\\
&\mathcal{D}_l^{eg} \models Poss(repPr(c, x), s) \supset EnvTurn_{LL}(s)
\end{aligned}$$

□

Here, we will restrict our attention to bounded-length strategies.[34] Let us discuss how we can represent such strategies. We define *(bounded) strategies* γ as a restricted form of program using the following BNF rule:

$$\gamma ::= nil \mid [\alpha^o; \gamma] \mid \mathbf{set}(P_{ES})$$

where P_{ES} is a non-empty set of programs of the form $[\beta_i^e; \gamma_i]$.

In the above, nil is a special program, called the *empty program*, that denotes the fact that nothing remains to be performed. α^o ranges over ordinary primitive action terms, β_i^e ranges over exogenous primitive action terms, and γ_i over strategies where $i \in \{1, 2, \ldots\}$. Thus nil is the strategy that does nothing, $[\alpha^o; \gamma]$ represents the strategy where the agent does action α^o and then follows strategy γ, and $\mathbf{set}([\beta_1^e; \gamma_1], [\beta_2^e; \gamma_2], \ldots)$ with distinct actions β_1^e and β_2^e represent the strategy where exogenous action β_1^e may occur after which γ_1 is

[34]For more general types of tasks that require unbounded strategies, it is possible to use approaches similar to [139, 141].

followed, exogenous action β_2^e may occur after which γ_2 is followed, etc. There may be a finite or countably infinite set of such pairs $(\beta_i^e; \gamma_i)$.

Informally, an agent is *able/knows how* to execute a task/program δ in a situation $do(\vec{a}, S_0)$ if whenever it is her turn to act, she is able to choose some action that she knows to be executable and is allowed by her program, such that no matter what exogenous actions occur (as allowed by the program), she can continue this process with what remains of the program and eventually reach a configuration where she knows that she can legally terminate.

We formalize $AbleBy(\delta, \vec{a}, \gamma)$ meaning that the agent *is able to successfully perform a task* represented by an *online situation-determined program* δ in an environment that behaves as specified by δ in situation $do(\vec{a}, S_0)$ by executing the strategy γ. Formally, similarly to [99], let $AbleBy(\delta, \vec{a}, \gamma)$ be the smallest relation $R(\delta, \vec{a}, \gamma)$ such that:

(A) for all pairs $(\delta, \vec{a})$, if $(\langle \delta, \vec{a} \rangle)^{\checkmark}$, then $\mathcal{R}(\delta, \vec{a}, nil)$;

(B) for all $\delta, \vec{a}$, if there exists a, δ' such that $a \in \mathcal{A}^o$ and $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$ and $\mathcal{R}(\delta', \vec{a}a, \gamma)$, then $\mathcal{R}(\delta, \vec{a}, [a; \gamma])$;

(C) for all $\delta, \vec{a}$, if there exists a, δ' such that $a \in \mathcal{A}^e$ and $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$
and for all a, δ' there exists γ such that $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$ and $\mathcal{R}(\delta', \vec{a}a, \gamma)$
then $\mathcal{R}(\delta, \vec{a}, \mathbf{set}(E))$
where $E = \{[a; \gamma] \mid \exists \delta'$ such that $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$ and $R(\delta', \vec{a}a, \gamma)\}$

Thus, for all δ and $\vec{a}$ we have that: (A) if δ is final in situation $do(\vec{a}, S_0)$, then the agent is able to execute δ in situation $do(\vec{a}, S_0)$ by performing the empty strategy (nil); (B) if there is an ordinary action a which is online executable at situation $do(\vec{a}, S_0)$ and thus the agent can make a transition to configuration $\langle \delta', \vec{a}a \rangle$ for some remaining program δ' where the agent is able to execute δ' in situation $do(\vec{a}a, S_0)$ by following strategy γ, then a is prefixed to the existing strategy γ and the agent is able to execute δ in situation $do(\vec{a}, S_0)$ by following strategy $([a; \gamma])$; (C) if there is an exogenous action a which is online executable at situation $do(\vec{a}, S_0)$ and thus the agent can make a transition to configuration $\langle \delta', \vec{a}a \rangle$ for some remaining program δ', and for all such a and δ' there exists a strategy γ such that the agent is able to execute δ' in situation $do(\vec{a}a, S_0)$ by following strategy γ, then at situation $do(\vec{a}, S_0)$, the agent is able to execute δ in situation $do(\vec{a}, S_0)$ by following strategy $\mathbf{set}(E)$, where E includes a sub-strategy $[a; \gamma]$ for each exogenous action a that may occur and γ is the strategy to follow after a occurs.

AbleBy provides a way that we can ensure the agent's task can be successfully completed. Note that the task may be simply achieving a given goal eventually by performing any sequence of actions. This can be represented by the program $Achieve(\phi) \doteq (\pi a.a)^*; \phi?$.

Example 7.9 Going back to our running example, suppose that the high-level agent has to perform the task specified by the online situation-determined program $\delta_{h_1} = \mathit{Achieve}(\phi_{h1})$, where $\phi_{h1} = (\mathit{Upg}(C1) \wedge \mathit{Booked}(C1, Bz, F1)) \vee (\neg \mathit{Upg}(C1) \wedge \mathit{Booked}(C1, Ec, F1))$. This task requires the agent to book a business seat for customer $C1$ on flight $F1$ if an upgrade is available and otherwise to book an economy seat. The high-level agent is able to achieve this task by executing the following strategy:

$\gamma_{h_1} = \mathit{qryUpg}(C1); \mathbf{set}((\mathit{repUpg}(C1,1); \mathit{book}(C1, Bz, F1); \mathit{nil}), (\mathit{repUpg}(C1,0); \mathit{book}(C1, \mathit{Eco}, F1); \mathit{nil}))$
Thus we have $\mathit{AbleBy}(\delta_{h_1}, \epsilon, \gamma_{h_1})$.

Referring back to Example 7.7, the task of achieving the goal of booking a business seat on flight $F1$ for customer $C1$ is specified by the online situation-determined program $\delta_{h_0} = \mathit{Achieve}(\phi_{h0})$, where $\phi_{h0} = \mathit{Booked}(C1, Bz, F1)$. In this case, there is no strategy γ such that $\mathit{AbleBy}(\delta_{h_0}, \epsilon, \gamma)$ since no upgrade may be available, i.e., $\mathcal{D}_h \cup \{\neg \mathit{Upg}(C1, S_0)\}$ is satisfiable. $\square$

Note that for $\mathit{AbleBy}(\delta, \vec{a}, \gamma)$ to hold, the agent's strategy γ need only ensure that the task δ can be successfully completed when the environment chooses to perform (any) exogenous actions that are allowed by δ. That is, δ can restrict what the environment may do, as well as set objectives for the agent. Of course, we may define δ so that it leaves the environment completely unconstrained, i.e, so that it satisfies the following:

$$\begin{array}{l} \mathit{EnvUnconstrained}(\delta, s) \doteq \\ \quad \bigwedge_{A \in \mathcal{A}^e} \forall s', \delta', \vec{x}. \mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Poss}(A(\vec{x}), s') \supset \exists \delta''. \mathit{Trans}^*(\delta, s, \delta'', do(A(\vec{x}, s'))) \end{array}$$

7.4 Hierarchical Contingent Planning

The notion of strategy formalized in the previous section can be used by both the high-level and low-level agents to ensure successful execution of the program at each level.

If we have a strategy at the high level that ensures performing a task or achieving a goal, are there conditions under which we can obtain a strategy at the low level that is a refinement of the high-level strategy and can ensure performing refinement of the high-level task or achieving refinement of the high-level goal? What are the assumptions that we need?

7.4.1 Assumptions

First, it is reasonable to assume that Assumption 5.2 holds. Without this assumption, at the concrete level, there is nothing preventing the environment from performing an action that is not part of any refinement of any high-level action when it is its turn. Thus, if we have a strategy at the high level that achieves a goal/performs a task, it becomes irrelevant and it is impossible to realize this strategy at the low level; in this way, we can't use reasoning at the high level to guide the reasoning at the low level.

Example 7.10 Referring to our running example, the preconditions of each action impose a certain order in which the sequence of low-level actions can be executed that matches refinements of high-level actions; moreover, the preconditions ensure that once the first action in the sequence of actions that refine a high-level action has been performed, no action that is not part of the refinement of the high-level action can be interleaved. Thus, Assumption 5.2 is satisfied. $\square$

Secondly, we need to avoid cases where a transition exists at the high level, while the low level blocks, as no transition of a refinement of a high level action is executable.[35] Hence, we assume that after any sequence of refinements of high-level actions, i.e., in any situation s such that $Do(\textsc{anySeqHLRef}, S_0, s)$, a refinement of either an ordinary action or an exogenous action can successfully be executed:

Assumption 7.2 (Non-Blocking)

$$\begin{aligned}\mathcal{D}_l \cup \mathcal{C} \models \forall s. Do(\textsc{anySeqHLRef}, S_0, s) \supset \\ \bigvee_{A_i \in \mathcal{A}^o_{hl}} \exists \vec{x}\, \exists s'. Do(m(A_i(\vec{x})), s, s') \vee \\ \bigvee_{A_i \in \mathcal{A}^e_{hl}} \exists \vec{x}\, \exists s'. Do(m(A_i(\vec{x})), s, s')\end{aligned}$$

where $\mathcal{A}^o_{hl}$ (resp. $\mathcal{A}^e_{hl}$) represents the high-level ordinary (resp. exogenous) set of action types.

Example 7.11 Going back to our running example, we can show that Assumption 7.2 holds for all situations s such that $Do(\textsc{anySeqHLRef}, S_0, s)$. For example, at S_0, at the abstract level, no actions have been executed so far and thus, based on the initial theory, the fluent $\neg QrdUpg(C1, S_0)$ holds and $\neg EnvTurn_{HL}(S_0)$ is satisfied; thus action $qryUpg(C1)$ can be executed. Since $\mathcal{D}^{eg}_h$ is a sound abstraction of $\mathcal{D}^{eg}_l$ with respect to mapping m^{eg}, then at the corresponding situation at the low level, $\neg m(QrdUpg(C1, S_0))$, which refines to $\neg QrdAM(C1, S_0) \wedge \neg QrdPr(C1, S_0)$ also holds, and moreover, $\neg EnvTurn_{LL}(S_0)$ holds. In addition, the precondition of action $qryAM(C1)$ ensures that the process of execution of a refinement of any high-level action could not have already started. After preforming the $qryAM(C1)$, the fluent $QrdAM(C1, S_0)$ holds, and thus, the preconditions of the ordinary action $qryPr(C1)$ are also satisfied. Thus, the concrete agent can execute a refinement of the action $qryUpg(C1)$, which is $qryAM(C1); qryPr(C1)$. After performing $qryUpg(C1)$ at the high level, it is environment's turn to act, and a similar argument holds as in the previous case. Once either of the actions $repUpg(C1, 1)$ or $repUpg(C1, 0)$ have been executed at the high level, it is agent's turn to act, and she can either book a flight for customer $C1$ or perform the action

[35]Instead, we could assume the high-level theory is both a sound and *complete* abstraction of the low-level theory with respect to a mapping. However, this approach seems more restrictive; moreover, it would require showing that completeness is preserved as we execute the high-level strategy. We will investigate this approach in future work.

$qryUpg(c)$ for another customer. Again by a similar argument, a refinements of any of these actions can be executed at the concrete level. $\square$

We also need to rule out cases where the agent blocks because she does not know whether it is her turn to execute an action. Therefore, we assume that in any situation, both the high-level and low-level agents *know* whether it is their turn or the environment's turn; in other words, in any situation, it is *known* that only exogenous actions or only ordinary actions are executable:

Assumption 7.3 (Always Known Whose Turn It Is) *For $\mathcal{D} \in \{\mathcal{D}_h, \mathcal{D}_l\}$ and for all ground sequences $\vec{a}$ such that $\langle (\pi a.a)^*, \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \delta, \vec{a}\rangle$ and $\langle \delta, \vec{a}\rangle^{\checkmark}$ for some δ, we have that*

either
$$\mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \bigvee_{A^o \in \mathcal{A}} \exists \vec{x}.Poss(A(\vec{x}), s)$$
or
$$\mathcal{D} \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \bigvee_{A^e \in \mathcal{A}} \exists \vec{x}.Poss(A(\vec{x}), s)$$

Example 7.12 In our running example, at the abstract level at S_0, based on the fact that $\neg QrdUpg(C1)$ is known in the initial situation, and that $EnvTurn_{HL}(S_0)$ holds, the agent knows that it can perform a step online and execute $qryUpg(C1)$. After performing this action, we have that $\mathcal{D}_h^{eg} \cup \{Executable(do(qryUpg(C1), S_0))\} \models QrdUpg(C1) \wedge \neg RcvdRep(C1)$, thus the agent knows that $EnvTurn_{HL}(do(qryUpg(C1), S_0))$ holds and that it is environment's turn to act. Similarly, after execution $repUpg(C1, 1)$, we have that $\mathcal{D}_h^{eg} \cup \{Executable(do([qryUpg(C1), repUpg(C1, 1)], S_0))\} \models QrdUpg(C1) \wedge RcvdRep(C1) \wedge \neg Booked(C1, cls, f)$ (and similarly after execution of $repUpg(C1, 0)$ by the environment) which indicates that the agent knows that it is her turn to preform an action online. We can provide a similar argument for the low level. Thus, Assumption 7.3 is satisfied for both the high-level and low-level theories.

$\square$

We also need another assumption to ensure that the high-level theory remains a sound abstraction of the low-level theory with respect to a mapping, as a refinement of a high-level action is executed and the agent may obtain new knowledge. Therefore, we assume that along all online executions, whenever a refinement of a high-level action has been executed, the low-level agent *knows* that it has executed it:

Assumption 7.4 (Awareness of Executed HL Actions) *For all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle m(\vec{\alpha}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a}\rangle$ and $\langle \delta_l, \vec{a}\rangle^{\checkmark}$ for some δ_l, we have that for any ground high-level action β and any ground low-level action sequence $\vec{b}$, if $\langle m(\beta), \vec{a}\rangle \rightarrow^*_{\vec{b}} \langle \delta_l', \vec{a}\vec{b}\rangle$ and $\langle \delta_l', \vec{a}\vec{b}\rangle^{\checkmark}$ for some δ_l', then $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}\vec{b}, S_0))\} \models Do(m(\beta), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$.*

As discussed in Example 7.2 our running example satisfies the above assumption.

We also need to ensure that if it is known that there exists a refinement of an ordinary high-level action β that is executable at the low level then the agent has a strategy to successfully execute a refinement of β, no matter what the environment does. Let us look at an example:

Example 7.13 Suppose that we have three high-level ordinary actions: a_h and c_h which are always executable, and b_h which is executable in a situation after a_h has been performed, indicated by fluent $PerformedAh$; thus we have $Poss(b_h, s) \equiv PerformedAh(s)$. Assume that after performing b_h, the fluent $Goal_h$ holds and only then. The refinements of the high-level actions and fluents are defined as: $m(a_h) = a_l; e_1$, $m(b_h) = b_l$, $m(c_h) = a_l; e_2$, $m(PerformedAh) = PeformedAlE1$, and $m(Goal_h) = Goal_l$. At the concrete level, a_l is an ordinary action that is always executable, e_1 and e_2 are exogenous actions that are executable in a situation after a_l has been performed, indicated by the fluent $PeformedAl$, and b_l is an ordinary action that is executable in the situation after e_1 has been performed. $Goal_l$ becomes true after the sequence of actions $a_l; e_1; b_l$ and only then.

The high-level task is represented by $\delta_{h_2} = Achieve(Goal_h)$, and we have that $AbleBy(\delta_{h_2}, \epsilon, \gamma_{h_2})$, where $\gamma_{h_2} = a_h; b_h$. However, there is no strategy γ_{l_2} such that the agent could achieve $Goal_l$, as after execution of a_l by the agent, the environment may perform e_2. □

Hence, we assume that in every configuration, the low-level agent has a strategy to execute any ordinary high-level action such that it has some executable refinement:

Assumption 7.5 (Ability to Execute Ordinary HL Actions) *For all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle m(\vec{\alpha}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a}\rangle$ and $\langle \delta_l, \vec{a}\rangle^{\checkmark}$ for some δ_l, we have that for any ground high-level action $\beta \in \mathcal{A}^o$, if $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \exists s. Do(m(\beta), do(\vec{a}, S_0), s)$, then there exists a low-level strategy γ_l such that $AbleBy(m(\beta), \vec{a}, \gamma_l)$.*

Example 7.14 Referring to our running example, for the ordinary high-level action $qryUpg(c)$, for any ground low-level action sequences $\vec{a}$ where $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \exists s. Do(m(qryUpg(c)), do(\vec{a}, S_0), s)$, there exists a low-level strategy γ_l such that $AbleBy(m(qryUpg(c)), \vec{a}, \gamma_l)$, where $\gamma_l = qryAM(c); qryPr(c)$, and both $qryAM(c)$ and $qryPr(c)$ are ordinary actions which can be executed by the agent. Similarly for the ordinary high-level action $book(c, cls, f)$, there exists a low-level strategy γ_l such that $AbleBy(m(book(c, cls, f)), \vec{a}, \gamma_l)$, where $\gamma_l = select(c, cls, f); pay(c, cls, f)$.

□

We further need to ensure that when a refinement of a high-level exogenous action is possibly executable, it should be the case that no matter what the agent or the environment do at the concrete level, a refinement of that exogenous action will eventually be successfully executed. To do this, similar to [98], we define a predicate $NecTerminates(\delta, \vec{a}) \doteq R(\delta, \vec{a})$ where $R(\delta, \vec{a})$ is the least relation such that:

(A) for all pairs $(\delta, \vec{a})$,
if $\langle \delta, \vec{a} \rangle^{\checkmark}$ and there does not exist a, δ' such that $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$,
then $R(\delta, \vec{a})$;

(B) for all pairs $(\delta, \vec{a})$,
if there exists a, δ' such that $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$ and for all a, δ' such that $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta', \vec{a}a \rangle$, $R(\delta', \vec{a}a)$,
then $R(\delta, \vec{a})$.

Note that *NecTerminates* is somewhat similar to the operator $\mathbf{AF}\phi$ in branching time logic CTL^* [27], meaning on all paths eventually ϕ.

Now let us define a low-level program that characterizes the refinements of high-level exogenous actions:

$$\text{ANYONEEXOHL} \doteq \mathbf{setp}(\{\pi\vec{x}.m(A_i(\vec{x})) \mid A_i \in \mathcal{A}_h^e\}),$$
i.e., do any exogenous high-level primitive action,

where **setp**() is the "delayed commitment" non-deterministic branch construct that ensures the resulting program of refinements of high-level exogenous actions is situation-determined (see Section 6.1.2).

Given the above definitions, we assume that along any online execution, if executability of a refinement of a high-level exogenous action is satisfiable at the low level, then the agent eventually completes the execution of a refinement of some high-level action at the low level:

Assumption 7.6 (Exogenous HL Actions Never Block) *For all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l, we have that if there exists a high-level action $\beta \in \mathcal{A}^e$ such that $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta), do(\vec{a}, S_0), s\}$ is satisfiable, then we have that NecTerminates*(ANYONEEXOHL, $\vec{a}$).

Example 7.15 Going back to our running example, the exogenous high-level action $repUpg(c, 0)$ is refined to sequence of exogenous low-level actions $repAM(c, 0); repPr(c, 0)$. Based on the preconditions for these actions, they can be performed when it is environment's turn and no other action can be interleaved with their execution. Similar argument holds for the exogenous high-level action $repUpg(c, 1)$. Thus, for the exogenous high-level action $repUpg(c, x)$, for any ground low-level action sequences $\vec{a}$ where $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(repUpg(c, x)), do(\vec{a}, S_0), s\}$ is satisfiable, then we have that *NecTerminates*(ANYONEEXOHL, $\vec{a}$). □

7.4.2 Results

Now assume that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m. We can show that if the high-level agent knows that she can execute an ordinary action β, then there exists a strategy γ_b at the low level by which the concrete agent can ensure to successfully execute a refinement of β:

Theorem 7.4 *Suppose that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m and Assumptions 7.1 and 7.5 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon \rangle \rightarrow^*_{\vec{\alpha}} \langle \delta'_h, \vec{\alpha} \rangle$ for some δ'_h and $\langle \delta'_h, \vec{\alpha} \rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l and for any ground high-level action $\beta \in \mathcal{A}^o$, if $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Poss(\beta, do(\vec{\alpha}, S_0))$, then there exists a low-level strategy γ_l such that $AbleBy(m(\beta), \vec{a}, \gamma_l)$.*

This result follows immediately from Theorem 7.2 and Assumption 7.5: when we have a sound abstraction, if an action is known to be executable at the high level, then at low level it is entailed that some refinement of it is executable by Theorem 7.2, and thus we can use Assumption 7.5 to get the result.

Moreover, if $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m, we can show that if executability of any exogenous action β is satisfiable at the high level, then at the low level, successful execution of a refinement of β is also satisfiable:

Theorem 7.5 *Suppose that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m and Assumptions 7.1, 7.2, 7.3, and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon \rangle \rightarrow^*_{\vec{\alpha}} \langle \delta'_h, \vec{\alpha} \rangle$ for some δ'_h and $\langle \delta'_h, \vec{\alpha} \rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l if there exists a ground high-level action $\beta \in \mathcal{A}^e$ such that $D_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta, do(\vec{\alpha}, S_0))\}$ is satisfiable, then $NecTerminates($ $\textsc{anyOneExoHL}, \vec{a})$.*

Now assume that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m. We can show that if the high-level agent has a strategy γ_h to successfully execute a task represented by δ_h, then there exists a low-level strategy γ_l such that the low-level agent can ensure successful execution of the refinement of γ_h by using strategy γ_l, a refinement of γ_h:

Theorem 7.6 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m and Assumptions 5.2, 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon \rangle \rightarrow^*_{\vec{\alpha}} \langle \delta'_h, \vec{\alpha} \rangle$ for some δ'_h and $\langle \delta'_h, \vec{\alpha} \rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l and for any online situation-determined high-level program δ_h and high-level strategy γ_h, if $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$, then there exists a low-level strategy γ_l such that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$.*

Note that here, we use the extended mapping m_p defined in Section 6.3 to map the high-level strategy γ_h to a low-level program that represents its refinements.

Under similar conditions as above, it also follows that if it is known at the high level that after successful execution of δ_h, a situation-suppressed formula ϕ holds, and that γ_h is a strategy that can ensure δ_h successfully terminates, then there exists a low-level strategy γ_l such that the low-level agent can ensure successful execution of the refinement of γ_h by using strategy γ_l, *after which the refinement of ϕ holds*:

Corollary 7.7 *Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m and Assumptions 5.2, 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon \rangle \rightarrow^*_{\vec{\alpha}} \langle \delta'_h, \vec{\alpha} \rangle$ for some δ'_h and $\langle \delta'_h, \vec{\alpha} \rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a} \rangle$ and $\langle \delta_l, \vec{a} \rangle^{\checkmark}$ for some δ_l and for any high-level online situation determined program δ_h, for any high-level strategy γ_h, and any situation-suppressed formula ϕ, if $\mathcal{D}_h \models Do(\delta_h, do(\vec{\alpha}, S_0), s') \supset \phi[s']$ and $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$ then there exists γ_l such that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$ and $AbleBy(m_p(\gamma_h); m(\phi)?, \vec{a}, \gamma_l)$.*

Example 7.16 Referring to our running example, as discussed in Example 7.9, our agent has the high-level strategy γ_{h_1} by which she can execute the task δ_{h1} and achieve the goal ϕ_{h1}. Based on Corollary 7.7, there exists a low-level strategy γ_{l_1} by which the low-level agent is able to accomplish a refinement of the goal $m(\phi_{h1})$; in fact we can take

$$\begin{aligned}\gamma_{l_1} = qryAM(C1); qryPr(C1); \mathbf{set}(&\\ &(repAM(C1,1); repPr(C1,1); selectFlt(C1,Bz,F1); pay(C1,Bz,F1); nil),\\ &(repAM(C1,1); repPr(C1,0); selectFlt(C1,Bz,F1); pay(C1,Bz,F1); nil),\\ &(repAM(C1,0); repPr(C1,1); selectFlt(C1,Bz,F1); pay(C1,Bz,F1); nil),\\ &(repAM(C1,0); repPr(C1,0); selectFlt(C1,Eco,F1); pay(C1,Eco,F1); nil))\end{aligned}$$

□

7.5 Discussion

In this chapter, we identified a sufficient property for a sound abstraction to persist along an online execution. We also showed results extending basic properties of sound abstractions to online executions (Theorem 7.2 and Proposition 7.3). We then adapted definitions of strategies and ability to perform a task/achieve a goal to our model of online execution. Based on this, we showed that under some reasonable assumptions, if we have a sound abstraction and the agent has a strategy by which she is able to perform a task/achieve a goal at the high level, then one can refine it into a low-level strategy by which the agent is able to perform/achieve the refinement of the task/goal. For simplicity, we focused on a single layer of abstraction, but the framework supports extending the hierarchy to more levels. Our approach can also support the use of ConGolog programs to specify the possible behaviors of the agent at both the high and low level, as we can follow [37] and "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.

In AI, Giunchiglia and Walsh [74] formalize abstraction as *syntactic* mappings between formulas of a concrete and a more abstract representation, while Nayak and Levy [122] present a *semantic* theory of abstraction. As discussed in Chapter 5, these approaches formalize abstraction of *static* logical theories, while our work focuses on abstraction of *dynamic* domains, where the theory may be updated as new knowledge becomes available during execution.

Several approaches have studied planning under incomplete information and sensing. These include PKS [126], a knowledge-based planner based on a generalization of STRIPS; planning based on model checking [26]; and a dynamic programming method for computing belief-based policies and a heuristic search method for computing history-based policies proposed by Geffner and Bonet [66]. These approaches do not consider abstraction.

Several notions of planning with abstraction have also been investigated. One approach is *precondition elimination abstraction*, first introduced in context of ABSTRIPS [135]. This work supports plans with information gathering operators by providing abstraction over the preconditions and effects of the operators, although some of the effects may have to be described in terms of uninstantiated parameters. This approach does not consider abstraction of actions. Another approach proposes *Hierarchical Task Networks* (HTNs) (e.g., [57]), which abstract over a set of (non-primitive) tasks. Encodings of HTNs in ConGolog with enhanced features like exogenous actions and online executions have also been studied by Gabaldon [63]. In contrast to our approach, [63] uses a single BAT; also it does not provide abstraction for fluents. Another approach is planning with *macro operators* (e.g., [89]), which represent meta-actions built from a sequence of action steps. McIlraith and Fadel [112] and Baier and McIlraith [10] investigate planning with *complex actions* (a form of macro actions) specified as Golog [103] programs. Baier and McIlraith [10] propose an offline execution semantics for Golog programs with sensing. Differently from our approach, [112, 10] compile the abstracted actions into a new BAT that contains both the original and abstracted actions. Also, they only deal with deterministic complex actions and do not provide abstraction for fluents. Moreover, our approach provides a refinement mapping between an abstract BAT and a concrete BAT.

Finally, Aguas et al. [1] propose hierarchical finite state controllers for *generalized planning* that can solve a range of similar planning problems. Hu and Levesque [86] propose representing generalized plan as a FSA plan [85] with its semantics defined in the situation calculus. Incomplete knowledge about the initial state is assumed, and the solution found can be used to solve multiple planning instances in the domain. Generalized planning is essentially different from our approach in that it focuses on abstracting over a solution for several instances, while we provide an abstraction for the problem first, and then use the solution found at the high level as guide for finding a solution at the low level. Another important difference between our work and the approaches focused on abstraction in planning is that they focus on improving the efficiency of planning, while our work provides a generic framework which can have applications in many areas.

8 Conclusion and Future Research

In this dissertation, we formalized frameworks for online agent supervision, hierarchical agent supervision, as well as general frameworks for abstraction of offline and online agent behavior. Although we approached the work on abstraction of offline and online agent behavior from the point of view of agent supervision, they have many applications beyond this area (e.g., monitoring, hierarchical planning, etc.). Our framework was based on the situation calculus [110, 132], a rich first-order logic language designed for representing and reasoning about dynamically changing worlds and the situation-determined variant of the ConGolog agent programming language [35]. While we studied these problems from a mainly theoretical perspective, we believe that our approach can support a range of practical tools (see Section 8.2 for a discussion). Control and customization of systems is appealing to various research communities, e.g., customization of software systems, behavior composition, and IoT, and the formalisms and techniques developed in this dissertation are amendable to such problems.

The next section provides an overview of our contributions. We conclude with some directions for future research in Section 8.2.

8.1 Summary of Contributions

The main contributions of this dissertation are as follows:

Online Agent Supervision. An agent executing online can acquire knowledge during a run, and at each time point she must make decisions on what to do next based on what her current knowledge is. We would like to be able to supervise such agents. To address this challenge, we first defined a notion of *online situation-determined agent* which ensures that for any sequence of actions that the agent can perform online, the resulting agent configuration (i.e., belief state and remaining program) is unique (Section 4.2.2). We then formalized the *online maximally permissive supervisor* (online MPS) and showed its existence and uniqueness (Section 4.3.2). Moreover, we meta-theoretically defined a *program construct* (i.e., supervision operator) for online supervised execution that given the agent and specification, executes them to obtain only runs allowed by the online maximally permissive supervisor, and we showed its soundness and completeness

(Section 4.3.3). To ensure the agent under the supervision operator construct considers only runs that can be successfully completed (i.e., ensure non-blockingness), we also defined a new lookahead search construct (Section 4.3.4). In the area of reasoning about knowledge and change, there is little work that considers customizing/controlling the behavior of an agent in presence of uncontrollable actions while leaving it as much autonomy as possible (e.g., [169, 44]). Similarly, in the literature on supervisory control of discrete event systems, there has been little focus on supervising online systems (e.g., [25, 76]). Our work, unlike these approaches, is based on an expressive first order logic framework.

Abstraction of Offline Agent Behavior. To facilitate reasoning about agents that exhibit complex behaviors, as well as to provide a high-level description of their behavior, we developed a general abstraction framework for agent behavior in offline executions. We formalized a notion of a high-level basic action theory being a *sound abstraction* of a low-level basic action theory under a given refinement mapping. This notion was based on a suitable notion of bisimulation between models of the high-level and low-level theories. We also provided a proof theoretic characterization that gives us the basis for automatically verifying that we have a sound abstraction (Section 5.3). In addition, we defined a dual notion of *complete abstraction* (Section 5.4). Moreover, we discussed how sound abstractions can be used to provide efficiency in planning. We also identified a set of constraints that ensure that for any low-level action sequence, there is a unique high-level action sequence that it refines. We discussed how this would be useful for providing high-level explanations of agent behavior and monitoring (Section 5.5). In first-order settings, most previous work on abstraction in dynamic domains has focused on specific applications such as hierarchical planning. Moreover, general frameworks for abstraction based on first-order logic have focused on static domains. Our approach on the other hand, provides a general abstraction framework in dynamic domains.

Hierarchical Agent Supervision. This work was motivated by use of hierarchies to make supervising complex agents more manageable, as due to the complexity of the behavior logic, designing and enforcing specifications for control/customization of agent's behavior can be difficult. We identified the constraints required to ensure that controllability of individual actions at the high level accurately reflected the controllability of their refinements. Then we showed that these constraints were in fact sufficient to ensure that any controllable set of runs at the high level had a controllable refinement that implements it and vice versa (Section 6.3). We also defined a new program construct that executes a set of programs P non-deterministically without committing to which element of P being executed unless it had to (Section 6.1.2). With the help of this construct and the constraints identified above we showed that the low-level MPS for the mapped specification was a refinement of the high-level maximally permissive supervisor for the specification (Section 6.3). Moreover, we showed that we could obtain the low-level MPS incrementally by using the high-level MPS as a guide and refining its actions locally while remaining maximally permissive. We then showed that the resulting hierarchically synthesized MPS had exactly the same runs as that of the low-level MPS obtained by mapping the supervision specification to the low level. We also argued that the hierarchically synthesized MPS would generally be much easier to compute compared to the low-level MPS obtained from the refined specification (Section 6.4). In the area of reasoning about actions and change, typical approaches that use abstraction are focused on specific applications, such as planning, and/or are not based on an expressive first order logic. Our approach was inspired by work in hierarchical supervisory control of discrete event systems [166]. The foundations of our work is different however: the framework was based on a rich first-order logic language; we used a notion of bisimulation to relate the models of the high-level and low-level theories; our high-level theory included fluents (which abstract over formulas) in addition to actions (that abstract over programs); and through preconditions for actions, we were able to enforce local constraints on the low-level agent.

Abstraction of Online Agent Behavior. To facilitate reasoning about agents with complex behaviors that may acquire new knowledge during a run, we developed a framework for abstraction of agent behavior in online executions. We identified a condition that ensured a high-level basic action theory remained a sound abstraction of a low-level basic action theory with respect to a refinement mapping as the agent acquired new knowledge (Section 7.2). We also formalized a model of contingent planning over agent's online executions that ensured that a strategy exists for the agent to only perform actions that could be extended to a successfully terminating execution of the program, no matter how the environment behaved (Section 7.3). We then showed that under some reasonable conditions, if we have sound abstraction and the agent has a conditional plan/strategy for accomplishing a task or achieving a goal at the high level, then we can refine it into a low-level strategy piecewise, and the resulting low-level strategy is guaranteed to achieve the refinement of the goal (Section 7.4). We also discussed how this approach could provide efficiency in contingent planning. While there has been previous work on contingent planning and hierarchical planning, there is little work that looks at both.

8.2 Further Research

There are a number of future directions for the work presented in this dissertation. We discuss some of them next. However, one may single out the topic of online hierarchical supervision of agents that was not addressed in this dissertation. This would be a very natural extension of our work that combines the results of online agent supervision and online abstraction of agent behavior with results of hierarchical agent supervision.

An important research direction for future work is developing methods and tools for solving interesting cases of the synthesis and verification problems for the notions formalized in this dissertation, i.e., synthesizing the most permissive supervisor, verifying that a BAT is a sound abstraction of another BAT with respect to a mapping, verifying that m-bisimilar models satisfy the local controllability assumption, synthesizing conditional plans using our abstraction framework, etc.

For verifying properties of basic action theories and ConGolog programs in the general infinite-states case, one can try to exploit general first-order and higher-order logic theorem proving techniques and tools; for instance, one could build upon Shapiro's work [150] that uses the PVS theorem proving system [124] to verify properties of situation calculus theories and ConGolog programs. In the case where the object domain is finite, then model checking techniques [9] could be adapted to perform the verification and synthesis problems that we are interested in. Another interesting case is when we have bounded basic action theories [36]. These are action theories where it is entailed that in all situations, the number of object tuples that belong to the extension of any fluent is bounded, although the object domain remains infinite and an infinite run may involve an infinite number of objects. It was shown that verifying μ-calculus properties over such theories is decidable (by showing that one can construct an abstract transition system that is bisimilar to infinite transition system that models infinite objects). Although no tools yet have been built to support verification in this setting, it could provide an exciting foundation for solving the problems that formalized. Finally, it would be interesting to identify cases where we can obtain decidability/complexity results for the problems formalized in this thesis. Related work in this area includes [30, 80].

For each of the main problems addressed in this dissertation, here are some topics for further research:

Online Agent Supervision

- Implementing frameworks and developing practical tools is an important topic for future work. If the object domain is finite then an implementation can be readily obtained by adapting discrete event system synthesis techniques [129]. On the other hand, if the object domain is infinite, one can look at bounded theories for implementation.
- Another direction for future research is to examine how we can relax the assumption that the supervisor and supervised agent share the same belief state.

Abstraction of Offline Agent Behavior

- In future work, one could investigate methodologies for designing abstract agents/theories and refinement mappings with respect to given objectives, as well as automated synthesis techniques to support such methodologies.
- One could also explore how using different types of mappings and basic action theories from various sources that yield sound/complete abstractions can support system evolvability.

- Investigating abstraction of domain entities is another possible direction for further research.
- One could also explore how agent abstraction can be used in verification of (partial) correctness of agent theories or programs; related work in this area includes application of predicate abstraction in verification of partial correctness of Golog programs [118] and infinite-state multi-agent systems [17].

Hierarchical Agent Supervision

- Similar to online agent supervision, one direction for future research is implementing frameworks and developing practical tools by adapting discrete event system synthesis techniques or methods based on bounded action theories.
- An interesting future direction would be to explore how "compatible" low-level specifications (i.e., those that do not cause the system to block due to inconsistencies with high-level specifications) on the concrete agent behavior can also be handled as we synthesize the low-level MPS.
- Moreover, one could 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.
- Furthermore, one could explore supervision of complex multi-agent systems, where different supervisors control individual agents/teams of agents forming various architectures, while ensuring that the entire system satisfies its specifications.

Abstraction of Online Agent Behavior

- A possible extension is to investigate conditions for persistence of complete abstractions in online executions, as well as the constraints that allow us to use hierarchical contingent planning in such settings.
- One could also explore how to verify whether the assumptions that are required for the case of hierarchical contingent planning that we developed hold.
- Similar to abstraction of offline agent behavior, some other further research in the online execution setting include investigating methodologies for designing abstract agents and refinement mappings with respect to given objectives in addition to automated synthesis techniques that support this, exploring how agent abstraction can be used in verification, and investigating abstraction of terms.

A Proofs

A.1 Online Agent Supervision

A.1.1 Some Results about the set Construct

In this section, we start by showing some results about the **set** construct that are used in proofs of the main results in Online Supervision.

Proposition A.1 *If* $\langle \mathbf{set}(E), \vec{a}\rangle \rightarrow_a c$ *and* $\langle \mathbf{set}(E), \vec{a}\rangle \rightarrow_a c'$*, then* $c = c'$*.*

Proof If a is not an exogenous action, then it must be known in $do(\vec{a}, S_0)$ that a is executable and there must be some action sequence in E that starts with a. The unique new configuration c is then $\langle \mathbf{set}(E'), \vec{a}a\rangle$ with $E' = \{\vec{b} \mid a\vec{b} \in E\}$. If a is an exogenous action, then it must be consistent in $do(\vec{a}, S_0)$ that a is executable and some action sequence in E must start with a; the unique new configuration c is just as in the previous case. $\square$

Then Corollary A.2 trivially follows. This corollary is used in proof of Theorem 4.2.

Corollary A.2 *Any agent* $\langle \mathcal{D}, \delta^i\rangle$ *with the initial program* $\delta^i = \mathbf{set}(E)$ *is online situation determined.*

The result of Lemma A.3 is used in proof of Theorem 4.2:

Lemma A.3 *If* $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \mathbf{set}(E)\rangle)$*, then there exists* $\vec{b}$ *such that* $\vec{a}\vec{b} \in E$*.*

Proof (Sketch) By induction on the length of $\vec{a}$. $\square$

The result of Lemma A.4 is used in proof of Lemma A.5:

Lemma A.4 *If* $\langle \delta^i, \epsilon\rangle \rightarrow^*_{\vec{a}} c$*, then* $\langle \mathbf{set}(E \cup \{\vec{a}\vec{b}\}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \mathbf{set}(E' \cup \{\vec{b}\}), \vec{a}\rangle$*.*

Proof (Sketch) By induction on the length of $\vec{a}$, noticing that the antecedent implies that the agent actions are known to be executable and the exogenous actions are thought to be possibly executable, and so the transitions exist. $\square$

The result of Lemma A.5 is used in proof of Theorem 4.2:

Lemma A.5
If $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \delta^i\rangle)$ *and* $\vec{a} \in E$*, then* $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \mathbf{set}(E)\rangle)$*.*

Proof Assume that the antecedent. Since $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \delta^i\rangle)$, there exists c such that $\langle \delta^i, \epsilon\rangle) \rightarrow^*_{\vec{a}} c$. Since $\vec{a} \in E$, it then follows by Lemma A.4 that $\langle \mathbf{set}(E), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \mathbf{set}(E'), \vec{a}\rangle$ with $\epsilon \in E'$. Thus $Final(\langle \mathbf{set}(E'), \vec{a}\rangle)$, and therefore $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \mathbf{set}(E)\rangle)$. $\square$

A.1.2 Online Situation-Determined Agents Proofs

Theorem 4.1 For any agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$, if δ^i is known to be SD in $\mathcal{D}$, i.e., $\mathcal{D} \cup \mathcal{C} \models \mathit{SituationDetermined}(\delta^i, S_0)$, and if σ always knows the remaining program after an exogenous action, then σ is online SD.

Proof By induction on the length of the action (and online transition) sequence. If the sequence is empty, the result trivially follows. Assume that the result holds for all action sequences of length k (IH). Suppose that $c^i \rightarrow^*_{\vec{a}a} \langle \delta_1, \vec{a}a \rangle$ and $c^i \rightarrow^*_{\vec{a}a} \langle \delta_2, \vec{a}a \rangle$ and $\delta_1 \neq \delta_2$ with $\vec{a}a$ of length $k+1$. It follows by the definition of online execution and the IH that $c^i \rightarrow^*_{\vec{a}} \langle \delta, \vec{a} \rangle$ and $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta_1, \vec{a}a \rangle$ and $\langle \delta, \vec{a} \rangle \rightarrow_a \langle \delta_2, \vec{a}a \rangle$ and $\delta_1 \neq \delta_2$. If action a is not exogenous, then by the definition of online transition $\mathcal{D} \cup \mathcal{C} \cup \{\mathit{Executable}(do(\vec{a}, S_0))\} \models \mathit{Trans}(\delta, a, \delta', do(\vec{a}, S_0))$ (where δ' is unique), which contradicts $\delta_1 \neq \delta_2$. If action a is exogenous, then both δ_1 and δ_2 are satisfiable as remaining programs. In each case, $Poss(a, do(\vec{a}, S_0))$ is also satisfiable.

Since σ always *knows* the remaining program after an exogenous action, we have that $\mathcal{D} \cup \mathcal{C} \cup \{\mathit{Executable}(do(\vec{a}a, S_0))\} \models \mathit{Trans}(\delta, a, \delta', do(\vec{a}, S_0))$ (where δ' is unique), which contradicts $\delta_1 \neq \delta_2$.
$\square$

A.1.3 Online MPS Proofs

Theorem 4.2 For the online maximally permissive supervisor $mps_{onl}(\delta^s, \sigma)$ of the online SD agent $\sigma = \langle \mathcal{D}, \delta^i \rangle$ which fulfills the supervision specification δ^s, where $\langle \mathcal{D}, \delta^s \rangle$ is also online SD, the following properties hold:

1. $mps_{onl}(\delta^s, \sigma)$ always exists and is unique;
2. $\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle$ is online SD;
3. $mps_{onl}(\delta^s, \sigma)$ is online controllable with respect to σ;
4. for every possible online controllable supervision specification $\hat{\delta^s}$ for σ such that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \& \hat{\delta^s} \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \& \delta^s \rangle)$, we have that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \& \hat{\delta^s} \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$, i.e., mps_{onl} is maximally permissive;
5. $\mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle) = \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$, i.e., $mps_{onl}(\delta^s, \sigma)$ is non-blocking.

Proof

Claim 1. The online MPS exists as $\mathbf{set}(\emptyset)$ satisfies the conditions to be included in $mps_{onl}(\delta^s, \sigma)$. Uniqueness follows from the existence of a supremal element.

Claim 2. Trivially follows from Corollary A.2.

Claim 3. It suffices to show that for all $\vec{a}$ and a_u such that $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$ and $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a}, S_0))\} \not\models \neg A^u(a_u, do(\vec{a}, S_0))$, we have that if $\vec{a}a_u \in \mathcal{GR}(\sigma)$ then $\vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$. Indeed, if $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$ then there is an online controllable supervision specification $\mathbf{set}(E)$ such that $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \mathbf{set}(E) \rangle)$. $\mathbf{set}(E)$ being online controllable wrt σ, if $\vec{a}a_u \in \mathcal{GR}(\sigma)$ then $\vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, \mathbf{set}(E) \rangle)$, but then $\vec{a}a_u \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$.

Claim 4. This follows immediately from the definition of $mps_{onl}(\delta^s, \sigma)$, by noticing that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \& \hat{\delta^s} \rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^i \& \mathbf{set}(E_{\hat{\delta^s}}) \rangle)$, and observing that $mps_{onl}(\delta^s, \sigma)$ is essentially the union of such controllable $\mathbf{set}(E_{\hat{\delta^s}})$.

Claim 5. Suppose that $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$. By the definition of mps_{onl}, $mps_{onl}(\delta^s, \sigma) = \mathbf{set}(E)$ where $E \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ \delta^s \rangle)$. Since $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \mathbf{set}(E) \rangle)$, by Lemma A.3 there exists $\vec{b}$ such that $\vec{a}\vec{b} \in E$. Since $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ \delta^s \rangle)$, by Lemma A.5, we have that $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \mathbf{set}(E) \rangle)$. It then follows by the definition of $\mathcal{GR}$ that $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma) \rangle)$. $\square$

A.1.4 Supervision Operator Proofs

Theorem 4.3

1. If $\langle \mathcal{D}, \delta^s \rangle$ and $\langle \mathcal{D}, \delta^i \rangle$ are online SD, then so is $\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle$.

2. $\delta^i \&^{onl}_{A_u} \delta^s$ is online controllable with respect to $\langle \mathcal{D}, \delta^i \rangle$.

Proof

Claim 1. By induction on the length of the action (and online transition) sequence. If the sequence is empty, the result trivially follows. Assume that the result holds for all action sequences $\vec{a}$ of length k (IH). We need to show that the result holds for all action sequences $\vec{a}a$ of length $k+1$.

Assume $\langle \delta^i \&^{onl}_{Au} \delta^s, \vec{a} \rangle \rightarrow_a c'$ where $c' = \langle \delta^{i'} \&^{onl}_{Au} \delta^{s'}, \vec{a}a \rangle$. Due to the way online transition is defined for $\&^{onl}_{Au}$, configuration c' can only be reached if we have both $\langle \delta^i, \vec{a} \rangle \rightarrow_a \langle \delta^{i'}, \vec{a}a \rangle$ and $\langle \delta^s, \vec{a} \rangle \rightarrow_a \langle \delta^{s'}, \vec{a}a \rangle$.

Since both $\langle \mathcal{D}, \delta^s \rangle$ and $\langle \mathcal{D}, \delta^i \rangle$ are online SD, they both evolve to a unique configuration. The new configuration of $\langle \mathcal{D}, \delta^i \&^{onl}_{Au} \delta^s \rangle$ (i.e., c') is obtained from these two, so it must be unique if it exists.

Claim 2. We have to show that for all $\vec{a}$ and a^u, $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$ and $\mathcal{D} \cup \mathit{Executable}(do(\vec{a}, S_0)) \not\models \neg A^u(a^u, do(\vec{a}, S_0))$ implies if $\vec{a}a^u \in \mathcal{GR}(\sigma)$ then $\vec{a}a^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$.

Since, wlog we assume that $\langle \mathcal{D}, \delta^i \rangle$ and $\langle \mathcal{D}, \delta^s \rangle$ started with a common controllable action, we can write $\vec{a} = \vec{a'} a^c \vec{a^u}$, where $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a'}, S_0))\} \models \neg A_u(a^c, do(\vec{a'}, S_0))$ and $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a'}a^c, S_0)), A_u(\vec{a^u}, do(\vec{a'}a^c, S_0))\}$ is satisfiable. Let $\langle \delta^{i'}, \vec{a'} \rangle$ and $\langle \delta^{s'}, \vec{a'} \rangle$ denote the configurations reached by $\langle \delta^i, \epsilon \rangle$ and $\langle \delta^s, \epsilon \rangle$ after performing $\vec{a'}$ respectively; in other words: $\langle \delta^i, \epsilon \rangle \rightarrow^*_{\vec{a'}} \langle \delta^{i'}, \vec{a'} \rangle$ and $\langle \delta^s, \epsilon \rangle \rightarrow^*_{\vec{a'}} \langle \delta^{s'}, \vec{a'} \rangle$ By the fact that $\vec{a'} a^c \vec{a^u} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$, we know that there is a configuration such that $\langle \delta^{i'} \&^{onl}_{A_u} \delta^{s'}, \vec{a'} \rangle \rightarrow_{a^c} \langle \delta^{i''} \&^{onl}_{A_u} \delta^{s''}, \vec{a'}a^c \rangle$ But then by the definition of the online transition relation ($\rightarrow$) we have that for all $\vec{b^u}$ such that $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a'}a^c, S_0)), A_u(\vec{b^u}, do(\vec{a'}a^c, S_0))\}$ is satisfiable, if $\vec{a'}a^c\vec{b^u} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$ then $\vec{a'}a^c\vec{b^u} \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$. In particular this holds for $\vec{b^u} = \vec{a^u}a^u$. Hence we have that if $\vec{a}a^u \in \mathcal{GR}(\sigma)$ then $\vec{a}a^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$. □

Theorem 4.4

$$\mathcal{CR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle).$$

Proof We start by showing: $\mathcal{CR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$. By Theorem 4.3 claim 2 we have that $\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle$ is online controllable for $\langle \mathcal{D}, \delta^i \rangle$. Considering that $\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle$ is the largest online controllable supervisor for $\langle \mathcal{D}, \delta^i \rangle$, and that $\mathcal{RR}(\langle \mathcal{D}, \delta^i \,\&\, (\delta^i \&^{onl}_{A_u} \delta^s) \rangle) = \mathcal{RR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$, we get the thesis.

Next we prove: $\mathcal{CR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$. Suppose not. Then there exist a complete run $\vec{a}$ such that $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$ but $\vec{a} \notin \mathcal{CR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$. As an aside, notice that if $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \delta \rangle)$ then $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \delta \rangle)$, and for all prefixes $\vec{a'}$ such that $\vec{a'}\vec{b} = \vec{a}$, we have $\vec{a'} \in \mathcal{GR}(\langle \mathcal{D}, \delta \rangle)$. Hence, let $\vec{a'} = \vec{a''}a$ such that $\vec{a''} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$, $\vec{a''}a \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$, but $\vec{a''}a \notin \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$, and let $\langle \delta^i, \epsilon \rangle \rightarrow^*_{\vec{a''}} \langle \delta^{i''}, \vec{a''} \rangle$ and $\langle \delta^s, \epsilon \rangle \rightarrow^*_{\vec{a''}} \langle \delta^{s''}, \vec{a''} \rangle$. Since $\vec{a''}a \notin \mathcal{GR}(\langle \mathcal{D}, \delta^i \&^{onl}_{A_u} \delta^s \rangle)$, it must be the case that there is no configuration c such that $\langle \delta^{i''} \&^{onl}_{A_u} \delta^{s''}, \vec{a''} \rangle \rightarrow_a c$. Since, $\vec{a''}a \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$, it follows that both $\langle \delta^{i''}, \vec{a''} \rangle \rightarrow_a \langle \delta^{i'''}, \vec{a''}a \rangle$ and $\langle \delta^{s''}, \vec{a''} \rangle \rightarrow_a \langle \delta^{s'''}, \vec{a''}a \rangle$. But then it must be the case that $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a''}, S_0))\} \models \neg A_u(a, do(\vec{a''}, S_0))$, and there exists $\vec{b^u}$ such that $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a''}a, S_0)), A_u(\vec{b_u}, do(\vec{a''}a, S_0))\}$ is satisfiable and $\vec{a''}a\vec{b^u} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$ but $\vec{a''}a\vec{b^u} \notin \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$.

Notice that $\vec{b^u} \neq \epsilon$, since we have that $\vec{a''}a \in \mathcal{GR}(\langle \mathcal{D}\ , \delta^s \rangle)$. So $\vec{b^u} = \vec{c^u} b^u \vec{d^u}$ with $\vec{a''}a\vec{c^u} \in \mathcal{GR}(\langle \mathcal{D}\ , \delta^s \rangle)$ but $\vec{a''}a\vec{c^u}b^u \notin \mathcal{GR}(\langle \mathcal{D}\ , \delta^s \rangle)$. Now $\vec{a'} \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$ and $\mathcal{D} \cup \{\mathit{Executable}(do(\vec{a''}a, S_0)), A_u(\vec{c^u}b^u, do(\vec{a''}a, S_0))\}$ is satisfiable, we have that $\vec{a'}\vec{c^u}b^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \,\&\, \mathit{mps}_{onl}(\delta^s, \sigma) \rangle)$; this holds since,

$mps_{onl}(\delta^s, \sigma)$ is controllable for σ, and we have that, if $\vec{a'}\vec{c^u}b^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$ then $\vec{a'}\vec{c^u}b^u \in \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle)$. This, by the definition of $mps_{onl}(\delta^s, \sigma)$, implies $\vec{a'}\vec{c^u}b^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ \delta^s \rangle)$. Hence, we can conclude that $\vec{a'}\vec{c^u}b^u \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$, getting a contradiction. □

Theorem 4.5 If $\langle \mathcal{D}, \delta^s \rangle$ and $\langle \mathcal{D}, \delta^i \rangle$ are online SD, than so is $\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle$.

Proof If $\langle \mathcal{D}, \delta^s \rangle$ and $\langle \mathcal{D}, \delta^i \rangle$ are online SD, than so is $\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle$. By induction on the length of the action (and online transition) sequence. It can be shown in a similar way to Theorem 4.3 claim 1. □

Theorem 4.6 Suppose that we have an agent $\langle \mathcal{D}, \delta^i \rangle$, and a supervision specification δ^s which are online SD. Suppose also that δ^s is online controllable with respect to $\langle \mathcal{D}, \delta^i \rangle$, and that $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$. Then we have that:

1. $\mathcal{CR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle)$, i.e., the complete runs of $\Sigma^w_{onl}(\delta^s, \delta^i)$ are the complete runs of δ^s.
2. If $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \neq \emptyset$, then $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$, i.e., the partial runs of $\Sigma^w_{onl}(\delta^s, \delta^i)$ are the good runs of δ^s.
3. If $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \neq \emptyset$, then $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$, i.e., partial runs must be good runs, and the resulting program is "non blocking".

Proof

Claim 1. ($\subseteq$) Suppose that $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \Sigma^w_{ol}(\delta^s, \delta^i)\rangle)$. By the definition of online transition for Σ^w_{ol}, it is easy to show that $\langle \delta^s, \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \delta^{s'}, \vec{a} \rangle$ for some $\delta^{s'}$. By the definition of $Final$ for Σ^w_{ol}, we must have that $Final(\langle \delta^{s'}, \vec{a} \rangle)$. Thus $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle)$.

($\supseteq$) Suppose that $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle)$. Since $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$, we also have that $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$. Clearly, every prefix of $\vec{a}$ is in $\mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$ and $\mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$. By the definition of online transition for Σ^w_{onl}, it is easy to show that $\langle \Sigma^w_{onl}(\delta^s, \delta^i), \epsilon \rangle \rightarrow^*_{\vec{a}} \langle \Sigma^w_{onl}(\delta^{s'}, \delta^{i'}), \vec{a} \rangle$ for some $\delta^{s'}$ and $\delta^{i'}$. By the definition of $Final$ for Σ^w_{onl}, we must have that $Final(\langle \Sigma^w_{onl}(\delta^{s'}, \delta^{i'}), \vec{a} \rangle)$. Thus $\vec{a} \in \mathcal{CR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$.

Claim 2. ($\subseteq$) By contradiction. Suppose that $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$ but $\vec{a} \notin \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$. Then there exists $\vec{b}$, a, and $\vec{c}$ such that $\vec{a} = \vec{b}a\vec{c}$ and $\vec{b} \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$ and $\vec{b}a \notin \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$ (note that since $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \neq \emptyset$, we have that $\epsilon \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$). If a is not an exogenous action, then by the definition of online transition for Σ^w_{onl}, $\vec{b}a \notin \mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$, and thus $\vec{a} \notin \mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$, contradiction. Suppose that a is an exogenous action. Since δ^s is controllable wrt $\langle \mathcal{D}, \delta^i \rangle$, if $\vec{b}a \in \mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$, then $\vec{b}a \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$, contradiction.

($\supseteq$) Suppose that $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$. Then there exists $\vec{b}$ such that $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle)$. Since $\mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$, we also have that $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$. Clearly, every prefix of $\vec{a}\vec{b}$ is in $\mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$ and $\mathcal{GR}(\langle \mathcal{D}, \delta^i \rangle)$. Thus by the definition of online transition for Σ^w_{onl}, it is easy to show that $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$.

Claim 3. ($\subseteq$) Suppose that $\vec{a} \in \mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$. By Claim 2, $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \delta^s \rangle)$. Then by the definition of $\mathcal{GR}$, there exists $\vec{b}$ such that $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \delta^s \rangle)$. By Claim 1, it follows that $\vec{a}\vec{b} \in \mathcal{CR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$. Thus $\vec{a} \in \mathcal{GR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^s, \delta^i)\rangle)$.

($\supseteq$) Follows trivially from the definitions of $\mathcal{RR}$ and $\mathcal{GR}$. □

Theorem 4.7

$$\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^i \ \&^{onl}_{A_u} \ \delta^s, \delta^i)\rangle) = \mathcal{RR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle).$$

Proof By Theorem 4.3 Claim 2 we have that $\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle$ is online controllable for $\langle \mathcal{D}, \delta^i \rangle$. By Theorem 4.4, $\mathcal{CR}(\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$. Thus by the definition of &, it is easy to show that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \rangle)$. Therefore by Theorem 4.6 Claim 2, we have that $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^i \ \&^{onl}_{A_u} \ \delta^s, \delta^i)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle)$.

Theorem 4.4 says that $\mathcal{CR}(\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle) = \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$, and since a set of complete runs has a unique set of prefixes, it follows that $\mathcal{GR}(\langle \mathcal{D}, \delta^i \ \&^{onl}_{A_u} \ \delta^s \rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$. Thus $\mathcal{RR}(\langle \mathcal{D}, \Sigma^w_{onl}(\delta^i \ \&^{onl}_{A_u} \ \delta^s)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$.

It remains to show that $\mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle) = \mathcal{RR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$. By the definition of mps_{onl}, $\mathcal{CR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i \ \& \ \delta^s\rangle)$. Thus by the definition of &, it is easy to show that $\mathcal{CR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) \subseteq \mathcal{CR}(\langle \mathcal{D}, \delta^i\rangle)$. Then by the definition of $\mathcal{GR}$ and $\mathcal{CR}$, it follows that $\mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) \subseteq \mathcal{GR}(\langle \mathcal{D}, \delta^i\rangle)$. Thus by the definition of &, it is easy to show that $\mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle) = \mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle)$. By Theorem 4.2, we have that $\mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) = \mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle)$. Since $\mathcal{GR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) \subseteq \mathcal{GR}(\langle \mathcal{D}, \delta^i\rangle)$, it follows that $\mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle) \subseteq \mathcal{GR}(\langle \mathcal{D}, \delta^i\rangle)$. Therefore by the definition of &, it is easy to show that $\mathcal{RR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle) = \mathcal{RR}(\langle \mathcal{D}, mps_{onl}(\delta^s, \sigma)\rangle)$. Thus, $\mathcal{RR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle) = \mathcal{GR}(\langle \mathcal{D}, \delta^i \ \& \ mps_{onl}(\delta^s, \sigma)\rangle)$. $\square$

A.2 Abstracting Offline Agent Behavior

A.2.1 m-Bisimulation

Lemma 5.1 If $s_h \sim_m^{M_h,M_l} s_l$, then for any high-level situation-suppressed formula ϕ, we have that:

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

Proof By induction of the structure of ϕ. $\square$

Theorem 5.2 If $M_h \sim_m M_l$, then for any sequence of ground high-level actions $\vec{\alpha}$ and any high-level situation-suppressed formula ϕ, we have that

$$M_l \models \exists s' Do(m(\vec{\alpha}), S_0, s') \wedge m(\phi)[s'] \quad \text{if and only if} \quad M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)].$$

Proof By induction of the length of $\vec{\alpha}$, using Lemma 5.1. $\square$

A.2.2 Sound Abstraction

Theorem 5.4 Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any ground high-level action sequence $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, if $\mathcal{D}_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$, then $\mathcal{D}_l \cup \mathcal{C} \models \exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$.

Proof Assume that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m and that $D_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$. Take an arbitrary model M_l of $\mathcal{D}_l \cup \mathcal{C}$. Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m, there exists a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$. Since $\mathcal{D}_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$, we have that $M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$. Since $M_h \sim_m M_l$, there exist an m-bisimulation relation B between M_h and M_l such that $\langle S_0^{M_h}, S_0^{M_l} \rangle \in B$. It is easy to show by induction on the length of $\vec{\alpha}$ that there exists a situation S such that $M_l, v[s/S] \models Do(m(\vec{\alpha}), S_0, s)$ and that $\langle do(\vec{\alpha}, S_0)^{M_h}, S \rangle \in B$. From the latter and the fact that $M_h \models \phi[do(\vec{\alpha}, S_0)]$, it follows by Lemma 5.1 that $M_l, v[s/S] \models m(\phi)[s]$. M_l was an arbitrarily chosen model of $\mathcal{D}_l \cup \mathcal{C}$ and thus it follows that $\mathcal{D}_l \cup \mathcal{C} \models \exists s. Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$. $\square$

Corollary 5.5 If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, then for any sequence of ground high-level actions $\vec{\alpha}$ and for any high-level situation-suppressed formula ϕ, we have that

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s \forall s'. Do(m(\vec{\alpha}), S_0, s) \wedge Do(m(\vec{\alpha}), S_0, s') \supset (m(\phi)[s] \equiv m(\phi)[s'])$$

Proof By contradiction. Suppose that there exist M_l and v such that $M_l, v \models \mathcal{D}_l \cup \mathcal{C} \cup \{Do(m(\vec{\alpha}), S_0, s) \wedge Do(m(\vec{\alpha}), S_0, s') \wedge m(\phi)[s] \wedge \neg m(\phi)[s']\}$. Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, by Corollary 5.3 $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)] \wedge \neg\phi[do(\vec{\alpha}, S_0)]\}$ is satisfiable, a contradiction. $\square$

Theorem 5.7 If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, then for any sequence of ground high-level actions $\vec{\alpha}$ and for any ground high-level action β, we have that

$$\mathcal{D}_l \cup \mathcal{C} \models \exists s. Do(m(\vec{\alpha}\beta), S_0, s) \supset (\forall s. Do(m(\vec{\alpha}), S_0, s) \supset \exists s'. Do(m(\beta), s, s'))$$

Proof Take an arbitrary model M_l of $\mathcal{D}_l \cup \mathcal{C}$ and valuation v and assume that $M_l, v \models \exists s. Do(m(\vec{\alpha}\beta), S_0, s)$. It follows that there exists s_l such that $M_l, v[s/s_l] \models Do(m(\vec{\alpha}), S_0, s) \wedge \exists s'. Do(m(\beta), s, s')$. Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m, there exists a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$. Thus there exists an m-bisimulation relation B between M_h and M_l such that $\langle S_0^{M_h}, S_0^{M_l} \rangle \in B$. Then, it is easy to show by induction on the length of $\vec{\alpha}$ that since $M_l, v[s/s_l] \models Do(m(\vec{\alpha}), S_0, s) \wedge \exists s'. Do(m(\beta), s, s')$, we must have that $M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta, do(\vec{\alpha}, S_0))$. Take an arbitrary situation s_l' and suppose that $M_l, v[s/s_l'] \models Do(m(\vec{\alpha}), S_0, s)$. Then it follows by induction on the length of $\vec{\alpha}$ that $\langle s_h, s_l' \rangle \in B$. Since $M_h \models Poss(\beta, do(\vec{\alpha}, S_0))$, we must also have that $M_l, v[s/s_l'] \models \exists s'. Do(m(\beta), s, s')$. Since s_l' was chosen arbitrarily, it follows that $M_l, v \models \forall s. Do(m(\vec{\alpha}), S_0, s) \supset \exists s'. Do(m(\beta), s, s')$. $\square$

To prove Theorem 5.9 (and Theorem 5.13), we start by defining lemmas A.6 and A.7.

Lemma A.6 *If $M_h \models \mathcal{D}^h$ for some high level theory $\mathcal{D}^h$ and $M_l \models \mathcal{D}^l \cup \mathcal{C}$ for some low level theory $\mathcal{D}^l$ and $M_h \sim_m M_l$ for some mapping m, then*

(a) $M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$,

(b) $M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$
$\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[s] \equiv m(F_i(\vec{y}))[s']))$,

where $\phi^{Poss}_{A_i}(\vec{x})$ is the right hand side of the precondition axiom for action $A_i(\vec{x})$, and $\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x})$ is the right hand side of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using D^h_{ca}.

Proof By contradiction. Assume that M_h is a model of a high level theory $\mathcal{D}^h$ and M_l is a model of a low level theory $\mathcal{D}^l$ and $\mathcal{C}$ and $M_h \sim_m M_l$. Suppose that condition (a) does not hold. Then there exists a ground high level action sequence $\vec{\alpha}$, a ground low level situation term S, and a ground high level action $A_i(\vec{x})$ such that $M_l \models Do(m(\vec{\alpha}), S_0, S)$ and either (*) $M_l \models m(\phi^{Poss}_{A_i}(\vec{x}))[S]$ and $M_l \not\models \exists s'.Do(m(A_i(\vec{x})), S, s')$ or (**) $M_l \not\models m(\phi^{Poss}_{A_i}(\vec{x}))[S]$ and $M_l \models \exists s'.Do(m(A_i(\vec{x})), S, s')$. In case (*), by Theorem 5.2, since $M_h \sim_m M_l$, it follows that $M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi^{Poss}_{A_i}(\vec{x})[do(\vec{\alpha}, S_0)]$. Since $M_h \models \mathcal{D}^h_{Poss}$, we must also have that $M_h \models Poss(A_i(\vec{x}), do(\vec{\alpha}, S_0))$, and thus that $M_h \models Executable(do([\vec{\alpha}, A_i(\vec{x})], S_0))$. Thus by Theorem 5.2, $M_l \models Do(m(\vec{\alpha}), S_0, S) \wedge \exists s' Do(m(A_i(\vec{x})), S, s')$, which contradicts (*). Case (**) can be shown to to lead to a contradiction by a similar argument.

Now suppose that condition (b) does not hold. Then there exists a ground high level action sequence $\vec{\alpha}$, a ground high level action $A_i(\vec{x})$, and ground low-level situation terms S and S' such that $M_l \models Do(m(\vec{\alpha}), S_0, S) \wedge Do(m(A_i(\vec{x})), S, S')$ and either (*) $M_l \models m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[S]$ and $M_l \not\models m(F_i(\vec{y}))[S']$ or (**) $M_l \not\models m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[S]$ and $M_l \models m(F_i(\vec{y}))[S']$. In case (*), by Theorem 5.2, since $M_h \sim_m M_l$, it follows that $M_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x})[do(\vec{\alpha}, S_0)] \wedge Poss(A_i(\vec{x}), do(\vec{\alpha}, S_0))$. Since $M_h \models \mathcal{D}^h_{ssa}$, we must also have that $M_h \models F_i(\vec{y})[do([\vec{\alpha}, A_i(\vec{x})], S_0)]$. Thus by Theorem 5.2, $M_l \models Do(m(\vec{\alpha}), S_0, S) \wedge Do(m(A_i(\vec{x})), S, S') \wedge m(F_i(\vec{y}))[S']$, which contradicts (*). Case (**) can be shown to to lead to a contradiction by a similar argument. $\square$

The above lemma implies that if $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ wrt m, then $\mathcal{D}^l$ must entail the mapped high level successor state axioms and entail that the mapped conditions for a high level action to be executable (from the precondition axioms of $\mathcal{D}^h$) correctly capture the executability conditions of their refinements.

Lemma A.7 *Suppose that $M_h \models \mathcal{D}^h$ for some high level theory $\mathcal{D}^h$ and $M_l \models \mathcal{D}^l \cup \mathcal{C}$ for some low level theory $\mathcal{D}^l$ and m is a mapping between the two theories. Then if*

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

(b) $M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$

and ***(c)*** $M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$
$\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[s] \equiv m(F_i(\vec{y}))[s']))$,

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

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

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

Let us show that B is an m-bisimulation relation between M_h and M_l. We need to show that if $\langle s_h, s_l\rangle \in B$, then it satisfies the three conditions in the definition of m-bisimulation. We prove this by induction n, the number of actions in s_h.
Base case $n = 0$: We have already shown that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$, so condition 1 holds. By Lemma 5.1, it follows that $M_h, v[s/s_h] \models \phi_A^{Poss}(\vec{x})[s]$ if and only if $M_l, v[s/s_l] \models m(\phi_A^{Poss}(\vec{x}))[s]$ for any high-level primitive action type $A \in \mathcal{A}_h$. Thus by the action precondition axiom for A, $M_h, v[s/s_h] \models Poss(A(\vec{x}), s)$ if and only if $M_l, v[s/s_l] \models m(\phi_A^{Poss}(\vec{x}))[s]$. By condition (b), we have that $M_l, v[s/s_l] \models m(\phi_A^{Poss}(\vec{x}))[s]$ if and only if $M_l, v[s/s_l] \models \exists s'.Do(m(A(\vec{x})), s, s')$. Thus $M_h, v[s/s_h] \models Poss(A(\vec{x}), s)$ if and only if there exists s_l' such that $M_l, v[s/s_l, s'/s_l'] \models Do(m(A(\vec{x})), s, s')$. By the way B is defined, $\langle do([\vec{\alpha}, A(\vec{x})], S_0)^{M_h,v}, s_l'\rangle \in B$ if and only if $M_l, v[s/s_l, s'/s_l'] \models Do(m(A(\vec{x})), s_l, s_l')$. Thus conditions (2) and (3) hold for $\langle s_h, s_l\rangle$.
Induction step: Assume that if $\langle s_h, s_l\rangle \in B$ and the number of actions in s_h is no greater than n, then $\langle s_h, s_l\rangle$ satisfies the three conditions in the definition of m-bisimulation. We have to show that this must also hold for any $\langle s_h, s_l\rangle \in B$ where s_h contains $n+1$ actions. First we show that condition 1 in the definition of m-bisimulation holds. If $\langle s_h, s_l\rangle \in B$ and s_h contains $n+1$ actions, then due to the way B is defined, there exists a ground high level action sequence $\vec{\alpha}$ of length n and a ground high level action $A(\vec{c})$ such that $s_h = do(A(\vec{c}), do(\vec{\alpha}, S_0))^{M_h}$, $s_h' = do(\vec{\alpha}, S_0)^{M_h}$, $M_l, v[s/s_l'] \models Do(m(\vec{\alpha}), S_0, s)$, and $\langle s_h', s_l'\rangle \in B$. s_h' contains n actions so by the induction hypothesis, $\langle s_h', s_l'\rangle$ satisfies the three conditions in the definition of m-bisimulation, in particular $s_h' \sim_m^{M_h,M_l} s_l'$. By Lemma 5.1, it follows that $M_h, v[s/s_h'] \models \phi_{F,A}^{ssa}(\vec{y}, \vec{c})[s]$ if and only if $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$ for any high-level fluent $F \in \mathcal{F}_h$. Thus by the successor state axiom for F, $M_h, v[s/s_h'] \models F(\vec{y}, do(A(\vec{c}), s))$ if and only if $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$. By condition (c), we have that $M_l, v[s/s_l'] \models m(\phi_{F,A}^{ssa}(\vec{y}, \vec{c}))[s]$ if and only if $M_l, v[s/s_l] \models m(F(\vec{y}))[s]$. Thus $M_h, v[s/s_h'] \models F(\vec{y}, do(A(\vec{c}), s))$ if and only if $M_l, v[s/s_l] \models m(F(\vec{y}))[s]$. Therefore, $s_h \sim_m^{M_h,M_l} s_l$, i.e., condition 1 in the definition of m-bisimulation holds.
We can show that $\langle s_h, s_l\rangle$, where s_h contains $n+1$ actions, satisfies conditions 2 and 3 in the definition of m-bisimulation, by exactly the same argument as in the base case. □

With these lemmas in hand, we can prove our main result:
Theorem 5.9 $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ relative to mapping m if and only if

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

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

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

Proof
($\Rightarrow$) By contradiction. Assume that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m. Suppose that condition (a) does not hold, i.e., there exists $\phi \in \mathcal{D}_{S_0}^h$ such that $\mathcal{D}_{S_0}^l \cup \mathcal{D}_{ca}^l \cup \mathcal{D}_{coa}^l \not\models m(\phi)$. Thus there exists a model M_l' of $\mathcal{D}_{S_0}^l \cup \mathcal{D}_{ca}^l \cup \mathcal{D}_{coa}^l$ such that $M_l' \not\models m(\phi)$, and this model can be extended to a model M_l of $\mathcal{D}_l \cup \mathcal{C}$ such that $M_l \not\models m(\phi)$. Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m, there exists a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$. By Theorem 5.2, it follows that $M_h \not\models \phi$. Thus $\mathcal{D}_h \not\models \mathcal{D}_{S_0}^h$, contradiction.

Now suppose that condition (b) does not hold. Then there exists a model M_l of $\mathcal{D}_l \cup \mathcal{C}$ such that M_l falsifies condition (b). Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m, there exists a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$. But then by Lemma A.6, M_l must satisfy condition (b), contradiction.

We can prove that condition (c) must hold using Lemma A.6 by the same argument as for condition (b).

($\Leftarrow$) Assume that conditions (a), (b), and (c) hold. Take a model M_l of $\mathcal{D}^l \cup \mathcal{C}$. Let M_h be a model of the high level language such that

(i) M_h has the same object domain as M_l and interprets all object terms like M_l,

(ii) $M_h \models \mathcal{D}^h_{ca}$,

(iii) $M_h \models \Sigma$,

(iv) $M_h, v \models F(\vec{x}, do(\vec{\alpha}, S_0))$ if and only if $M_l, v \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(F(\vec{x}))[s]$ for all fluents $F \in \mathcal{F}^h$ and all ground high-level action sequences $\vec{\alpha}$.

(v) $M_h \models Poss(A(\vec{x}), do(\vec{\alpha}, S_0))$ if and only if $M_l \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge \exists s' Do(m(A(\vec{x})), s, s')$

It follows immediately that $M_h \models \Sigma \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$. By condition (iv) above, we have that $S_0^{M_h} \sim_m^{M_h, M_l} S_0^{M_l}$. Thus by condition (a) and Lemma 5.1, we have that $M_h \models \mathcal{D}^h_{S_0}$. By condition (b) of the Theorem and conditions (iv) and (v) above, $M_h \models \mathcal{D}^h_{Poss}$. By condition (c) of the Theorem and condition (iv) above, $M_h \models \mathcal{D}^h_{ssa}$. Thus $M_h \models \mathcal{D}^h$.
Now M_h and M_l satisfy all the conditions for applying Lemma A.7, by which it follows that $M_h \sim_m M_l$. $\square$

Theorem A.8 *$\mathcal{D}^{eg}_h$ is a sound abstraction of $\mathcal{D}^{eg}_l$ wrt m^{eg}.*

Proof We prove this using Theorem 5.9.
(a) It is easy to see that $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa} \models m(\phi)$, for all $\phi \in \mathcal{D}^h_{S_0}$ assuming that $\mathcal{D}^l_{S_0}$ entails all the facts about $CnRoute_{LL}$ that $\mathcal{D}^h_{S_0}$ contains.
(b) For the *deliver* high level action, we need to show that:

$$\begin{aligned}&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\ &\qquad \forall sID.(m(\exists l.Dest_{HL}(sID, l, s) \wedge At_{HL}(sID, l, s)) \equiv \exists s' Do(m(deliver(sID)), s, s')),\end{aligned}$$

i.e.,

$$\begin{aligned}&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\ &\qquad \forall sID.(\exists l.Dest_{LL}(sID, l, s) \wedge At_{LL}(sID, l, s) \equiv \exists s' Do([unload(sID); getSignature(sID)], s, s')).\end{aligned}$$

It is easy to check that the latter holds as $\exists l.Dest_{LL}(sID, l, s) \wedge At_{LL}(sID, l, s)$ is the precondition of $unload(sID)$ and $unload(sID)$ ensures that the precondition of $getSignature(sID)$.

For the *takeRoute* action, we need to show that:

$$\begin{aligned}&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \forall sID, r, o, d. \\ &\qquad (m(o \neq d \wedge At_{HL}(sID, o, s) \wedge CnRoute_{HL}(r, o, d, s) \wedge (r = Rt_B \supset \neg Priority(sID, s))) \\ &\qquad \equiv \exists s' Do(m(takeRoute(sID, r, o, d)), s, s'),\end{aligned}$$

i.e.,

$$\begin{aligned}&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \forall sID, r, o, d. \\ &\qquad o \neq d \wedge At_{LL}(sID, o, s) \wedge CnRoute_{LL}(r, o, d, s) \wedge \\ &\qquad (r = Rt_B \supset \neg(BadWeather(s) \vee Express(sID, s))) \\ &\qquad \equiv \exists s' Do(m(takeRoute(sID, r, o, d)), s, s').\end{aligned}$$

It is easy to show that the latter holds as the left hand side of the $\equiv$ is equivalent to $m(takeRoute(sID, r, o, d))$ being executable in s. First, we can see that the left hand side of the $\equiv$ is equivalent to the preconditions of first *takeRoad* action in $m(takeRoute(sID, r, o, d))$, noting that in the case where $r = Rt_B$,

takeRoute(sID, r, o, d) is mapped into *takeRoad* to destination $L3$. Moreover, the preconditions of the second *takeRoad* action in $m(takeRoute(sID, r, o, d))$ must hold given that the first *takeRoad* has occured and that there is some road that is not closed to go to d. The latter can be shown by induction on situations.
(c) For the high level action *deliver* we must show that:

$$\begin{aligned}
&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\
&\quad \forall sID, s'.(Do(m(deliver(sID)), s, s') \supset \\
&\quad \textstyle\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,deliver}(\vec{y}, sID))[s] \equiv m(F_i(\vec{y}))[s'])).
\end{aligned}$$

For the high level fluent *Delivered*, we must show that

$$\begin{aligned}
&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\
&\quad \forall sID, s'.(Do(m(deliver(sID)), s, s') \supset \\
&\quad \forall sID'(sID = sID' \equiv Unloaded(sID, s') \wedge Signed(sID, s')).
\end{aligned}$$

This is easily shown given that $m^{eg}(deliver(sID)) = unload(sID); getSignature(sID)$, using that successor state axioms for *Unloaded* and *Signed*. For the other high level fluents, the result follows easily as $m^{eg}(deliver(sID))$ does not affect their refinements.
For the action *takeRoute* we must show that:

$$\begin{aligned}
&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\
&\quad \forall sID, r, o, d, s'.(Do(m(takeRoute(sID, r, o, d)), s, s') \supset \\
&\quad \textstyle\bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,takeRoute}(\vec{y}, sID, r, o, d))[s] \\
&\qquad\qquad \equiv m(F_i(\vec{y}))[s'])).
\end{aligned}$$

For the high level fluent At_{HL}, we must show that

$$\begin{aligned}
&\mathcal{D}^l \cup \mathcal{C} \models Do(\text{ANYSEQHLREF}, S_0, s) \supset \\
&\quad \forall sID, r, o, d, s'.(Do(m(takeRoute(sID, r, o, d)), s, s') \supset \\
&\quad \forall sID', l.(At_{LL}(sID', l, s')) \equiv \\
&\qquad (sID' = sID \wedge l = d) \vee \\
&\qquad At_{LL}(sID, l, s) \wedge \neg(sID' = sID \wedge o = l)).
\end{aligned}$$

This is easily shown given how *takeRoute* is refined by m^{eg}, using that successor state axioms for At_{LL}. For the other high level fluents, the result follows easily as $m^{eg}(takeRoute(sID, r, o, d))$ does not affect their refinements. □

A.2.3 Complete Abstraction

Theorem 5.11 Suppose that $\mathcal{D}_h$ is a complete abstraction of $\mathcal{D}_l$ relative to mapping m. Then for any ground high-level action sequence $\vec{\alpha}$ and any high-level situation-suppressed formula ϕ, if $\mathcal{D}_l \cup \mathcal{C} \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$, then $\mathcal{D}_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$.

Proof Assume that $\mathcal{D}_h$ is a complete abstraction of $\mathcal{D}_l$ wrt m and that $\mathcal{D}_l \cup \mathcal{C} \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$. Take an arbitrary model M_h of $\mathcal{D}_h$. Since $\mathcal{D}_h$ is a complete abstraction of $\mathcal{D}_l$ wrt m, there exists a model M_l of $\mathcal{D}_l \cup \mathcal{C}$ such that $M_h \sim_m M_l$. Since $\mathcal{D}_l \cup \mathcal{C} \models \exists s.Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$, we have that for any v, there exists a situation S such that $M_l, v[s/S] \models Do(m(\vec{\alpha}), S_0, s) \wedge m(\phi)[s]$. Since $M_h \sim_m M_l$, there exist an m-bisimulation relation B between M_h and M_l such that $\langle S_0^{M_h}, S_0^{M_l}\rangle \in B$. It is easy to show by induction on the length of $\vec{\alpha}$ that $M_h \models Executable(do(\vec{\alpha}, S_0))$ and that $\langle do(\vec{\alpha}, S_0)^{M_h}, S\rangle \in B$. From the latter and the fact that $M_l, v[s/S] \models m(\phi)[s]$, it follows by Lemma 5.1 that $M_h, v \models \phi[do(\vec{\alpha}, S_0)]$. M_h was an arbitrarily chosen model of $\mathcal{D}_h$ and v was arbitrary, and thus it follows that $\mathcal{D}_h \models Executable(do(\vec{\alpha}, S_0)) \wedge \phi[do(\vec{\alpha}, S_0)]$.
□

Theorem 5.12 If $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ relative to mapping m, then $\mathcal{D}^h$ is also a complete abstraction of $\mathcal{D}^l$ with respect to mapping m if and only if for every model M_h of $\mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$, there exists a model M_l of $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa}$ such that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$.

Proof Assume that $\mathcal{D}^h$ is a sound abstraction of $\mathcal{D}^l$ wrt mapping m.
($\Rightarrow$) Suppose that $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ wrt mapping m. Take an arbitrary model of M_h of $\mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$. Clearly, M_h can be extended to satisfy all of $\mathcal{D}^h$ (justified by Reiter's relative satisfiability theorem for basic action theories [132]). Since $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ wrt m, by definition, there exists a model M_l of $\mathcal{D}^l \cup \mathcal{C}$ such that $M_l \sim_m M_h$. It follows by the definition of m-bisimulation that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$.
($\Leftarrow$) Suppose that for every model M_h of $\mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$, there exists a model M_l of $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa}$ such that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$. Take an arbitrary model M_h of $\mathcal{D}^h$. Since M_h is also a model of $\mathcal{D}^h_{S_0} \cup \mathcal{D}^h_{ca} \cup \mathcal{D}^h_{coa}$, then there exists a model M_l of $\mathcal{D}^l_{S_0} \cup \mathcal{D}^l_{ca} \cup \mathcal{D}^l_{coa}$ such that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$. Clearly, M_l can be extended to satisfy all of $\mathcal{D}^l$ (justified by Reiter's relative satisfiability theorem for basic action theories [132]). Moreover, M_l can be extended to satisfy $\mathcal{C}$ (by the results in [33]). Since $\mathcal{D}^h$ is also a sound abstraction of $\mathcal{D}^l$ wrt m, by Theorem 5.9 it follows that
$M_l \models Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$
and $M_l \models Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$
$\qquad \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[s] \equiv m(F_i(\vec{y}))[s']))$, where $\phi^{Poss}_{A_i}(\vec{x})$ is the right hand side of the precondition axiom for action $A_i(\vec{x})$, and $\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x})$ is the right hand side of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using $\mathcal{D}^h_{ca}$. Thus by Lemma A.7, it follows that $M_h \sim_m M_l$. Thus $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ wrt m, by definition of complete abstraction. $\square$

Theorem 5.13 $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ relative to mapping m iff for every model M_h of $\mathcal{D}^h$, there exists a model M_l of $\mathcal{D}^l \cup \mathcal{C}$ such that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$ and
$M_l \models \forall s.Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$
and $M_l \models \forall s.Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$
$\qquad \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[s] \equiv m(F_i(\vec{y}))[s']))$,
where $\phi^{Poss}_{A_i}(\vec{x})$ and $\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x})$ are as in Theorem 5.9.

Proof
($\Rightarrow$) Suppose that $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ wrt mapping m. Take an arbitrary model of M_h of $\mathcal{D}^h$. Since $\mathcal{D}^h$ is a complete abstraction of $\mathcal{D}^l$ wrt m, by definition, there exists a model M_l of $\mathcal{D}^l \cup \mathcal{C}$ such that $M_l \sim_m M_h$. It follows by the definition of m-bisimulation that $S_0^{M_h} \sim_m^{M_h,M_l} S_0^{M_l}$. Furthermore, by Lemma A.6, it follows that
$M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}.(m(\phi^{Poss}_{A_i}(\vec{x}))[s] \equiv \exists s' Do(m(A_i(\vec{x})), s, s'))$
and $M_l \models \forall s Do(\text{ANYSEQHLREF}, S_0, s) \supset$
$\quad \bigwedge_{A_i \in \mathcal{A}^h} \forall \vec{x}, s'.(Do(m(A_i(\vec{x})), s, s') \supset$
$\qquad \bigwedge_{F_i \in \mathcal{F}^h} \forall \vec{y}(m(\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x}))[s] \equiv m(F_i(\vec{y}))[s']))$,
where $\phi^{Poss}_{A_i}(\vec{x})$ is the right hand side of the precondition axiom for action $A_i(\vec{x})$, and $\phi^{ssa}_{F_i,A_i}(\vec{y},\vec{x})$ is the right hand side of the successor state axiom for F_i instantiated with action $A_i(\vec{x})$ where action terms have been eliminated using D^h_{ca}.
($\Leftarrow$) The thesis follows immediately from Lemma A.7 and the definition of complete abstraction. $\square$

A.2.4 Monitoring and Explanation

Theorem 5.14 For any refinement mapping m from $\mathcal{D}_h$ to $\mathcal{D}_l$, we have that:

1. $D_l \cup \mathcal{C} \models \forall s.\exists s'.lp_m(s, s')$,
2. $D_l \cup \mathcal{C} \models \forall s \forall s_1 \forall s_2.lp_m(s, s_1) \wedge lp_m(s, s_2) \supset s_1 = s_2$.

Proof
(1) We have that $\mathcal{D}_l \cup \mathcal{C} \models Do(\textsc{anyseqhlref}, S_0, S_0)$ since $\textsc{anyseqhlref}$ is a nondeterministic iteration that can execute 0 times. So even if there is no s'' such that $S_0 < s'' \leq s \wedge Do(\textsc{anyseqhlref}, S_0, s'')$, the result holds.
(2) Take an arbitrary model M_l of $\mathcal{D}_l \cup \mathcal{C}$ and assume that $M_l, v \models lp_m(s, s_1) \wedge lp_m(s, s_2)$. We have that $\mathcal{D}_l \cup \mathcal{C} \models lp_m(s, s') \supset s' \leq s$. Moreover, we have a total ordering on situations s' such that $s' \leq s$. If $M_l, v \models s_1 < s_2$, then s_1 can't be the largest prefix of s that can be produced by executing a sequence of high-level actions, and we can't have $M_l, v \models lp_m(s, s_1)$. Similarly if $M_l, v \models s_2 < s_1$, we can't have $M_l, v \models lp_m(s, s_2)$. It follows that $M_l, v \models s_1 = s_2$. $\square$

Theorem 5.15 Suppose that we have a refinement mapping m from $\mathcal{D}_h$ to $\mathcal{D}_l$ and that Assumption 5.1 holds. Let M_l be a model of $\mathcal{D}_l \cup \mathcal{C}$. Then for any ground situation terms S_s and S_e such that $M_l \models Do(\textsc{anyseqhlref}, S_s, S_e)$, there exists a unique ground high-level action sequence $\vec{\alpha}$ such that $M_l \models Do(m(\vec{\alpha}),\ S_s, S_e)$.

Proof Since, $M_l \models Do(\textsc{anyseqhlref}, S_s, S_e)$, there exists a $n \in \mathbb{N}$ such that $M_l \models Do(any1hl^n, S_0, S)$. Since we have standard names for objects, it follows that there exists a ground high-level action sequence $\vec{\alpha}$ such that $M_l \models Do(m(\vec{\alpha}), S_s, S_e)$. Now let's show by induction on the length of $\vec{\alpha}$ that there is no ground high-level action sequence $\vec{\alpha}' \neq \vec{\alpha}$ such that $M_l \models Do(m(\vec{\alpha}'), S_s, S_e)$. Base case $\vec{\alpha} = \epsilon$: Then $M_l \models Do(m(\vec{\alpha}), S_s, S_e)$ implies $M_l \models S_s = S_e$ and there is no $\vec{\alpha}' \neq \epsilon$ such that $M_l \models Do(m(\vec{\alpha}'), S_s, S_e)$, since by Assumption 5.1(c) $\mathcal{D}_l \cup \mathcal{C} \models Do(m(\beta), s, s') \supset s < s'$ for any ground high-level action term β. Induction step: Assume that the claim holds for any $\vec{\alpha}$ of length k. Let's show that it must hold for any $\vec{\alpha}$ of length $k + 1$. Let $\vec{\alpha} = \beta\vec{\gamma}$. There exists S_i such that $Do(m(\beta), S_s, S_i) \wedge S_i \leq S_e$. By Assumption 5.1(a), there is no $\beta' \neq \beta$ and S_i' such that $Do(m(\beta'), S_s, S_i') \wedge S_i' \leq S_e$. By Assumption 5.1(b), there is no $\mathcal{S}_i' \neq S_i$ such that $Do(m(\beta), S_0, S_i') \wedge S_i' \leq S$. Then by the induction hypothesis, there is no ground high-level action sequence $\vec{\gamma}' \neq \vec{\gamma}$ such that $M_l \models Do(m(\vec{\gamma}'), S_i, S_e)$. $\square$

Theorem 5.16 If m is a refinement mapping from $\mathcal{D}_h$ to $\mathcal{D}_l$ and Assumption 5.2 holds, then we have that:

$$\mathcal{D}_l \cup \mathcal{C} \models \forall s, s'.Executable(s) \wedge lp_m(s, s') \supset \exists \delta.Trans^* \textsc{any1hlref}, s', \delta, s)$$

Proof Take an arbitrary model M_l of $\mathcal{D}_l \cup \mathcal{C}$ and assume that $M_l, v \models Executable(s) \wedge lp_m(s, s')$. Since $M_l, v \models lp_m(s, s')$, we have that $M_l, v \models Do(\textsc{anyseqhlref}, S_0, s')$ and thus that $M_l, v \models Trans^*($ $\textsc{anyseqhlref}, S_0,\ \textsc{anyseqhlref}, s')$. Since $M_l, v \models Executable(s)$, by Assumption 5.2 we have that $M_l, v \models \exists \delta.Trans^*(\textsc{anyseqhlref}, S_0, \delta, s)$. Thus, it follows that $M_l, v \models \exists \delta.Trans^*(any1hl, s', \delta, s)$. $\square$

A.3 Hierarchical Agent Supervision

A.3.1 Hierarchical Controllability of High-Level Specifications

Lemma 6.1
$$\mathcal{D}_h \models \mathit{Controllable}(\mathbf{set}(E_S), \textsc{anyOne}, s) \equiv$$
$$\forall a_u . A^u_h(a_u, s) \wedge \mathit{Poss}(a_u, s) \supset \mathit{Do}(\mathbf{set}(E_S), s, \mathit{do}(a_u, s))$$

Proof
($\supset$) By contradiction. Take an arbitrary model M_h of $\mathcal{D}_h$, and assume the antecedent and the negation of the consequent. This means there is an action a_u such that $M_h \models A^u_h(a_u, s) \wedge \mathit{Poss}(a_u, s)$ and that $M_h \models \neg \mathit{Do}(\mathbf{set}(E_S), s, \mathit{do}(a_u, s))$. Since we have $M_h \models \mathit{Poss}(a_u, s)$ this implies that $a_u \in CR_{M_h}(\textsc{anyOne}, s)$. Since $M_h \models \neg \mathit{Do}(\mathbf{set}(E_S), s, \mathit{do}(a_u, s))$, this implies $a_u \notin CR_{M_h}(\mathbf{set}(E_S), s)$. But since $\mathit{Controllable}(\mathbf{set}(E_S), \textsc{anyOne}, s)$, we must have $a_u \in CR_{M_h}(\mathbf{set}(E_S), s)$, contradiction.

($\subset$) By contradiction. Take an arbitrary model M_h of $\mathcal{D}_h$, and assume the antecedent and the negation of the consequent. This means there is an action a_u such that $M_h \models A^u_h(a_u, s) \wedge \mathit{Poss}(a_u, s)$ and $a_u \in CR_{M_h}(\textsc{anyOne}, s)$ but $a_u \notin CR_{M_h}(\mathbf{set}(E_S), s)$. By the antecedent, we know that $a_u \in CR_{M_h}(\mathbf{set}(E_S), s)$, contradiction. $\square$

To show that Theorem 6.2 holds, we first need the following lemma about building controllable sets of runs out of controllable parts:

Lemma A.9 *For any program δ, any sequence of ground actions $\vec{a}$, and any set of ground action sequences E such that $E \subseteq \mathcal{CR}_M(\delta^*, \mathit{do}(\vec{a}, S_0))$ and $\epsilon \notin E$,*

if $M \models \mathit{Controllable}(\mathbf{set}(E^1), \delta, \mathit{do}(\vec{a}, S_0))$, where

$$E^1 = \{\vec{b} \mid \vec{b}\vec{c} \in E \text{ for some } \vec{c} \text{ and } M \models \mathit{Do}(\delta, \mathit{do}(\vec{a}, S_0), \mathit{do}(\vec{a}\vec{b}, S_0))\},$$

and for all $\vec{b} \in E^1$, $M \models \mathit{Controllable}(\mathbf{set}(E^{\vec{b}}), \delta^*, \mathit{do}(\vec{a}\vec{b}, S_0))$, where $E^{\vec{b}} = \{\vec{c} \mid \vec{b}\vec{c} \in E\}$,

then $M \models \mathit{Controllable}(\mathbf{set}(E), \delta^*, \mathit{do}(\vec{a}, S_0))$, and moreover,

if $M \models \mathit{Controllable}(\mathbf{set}(\{\epsilon\}), \delta^*, \mathit{do}(\vec{a}, S_0))$,
then $M \models \mathit{Controllable}(\mathbf{set}(E \cup \{\epsilon\}), \delta^*, \mathit{do}(\vec{a}, S_0))$.

Proof By contradiction. Assume the antecedent and the negation of consequent. The latter means that there are ground low-level action sequences $\vec{a'}\vec{e}$ and $\vec{a'}a_u\vec{d}$ such that $\vec{a'}\vec{e} \in \mathcal{CR}_M(\mathbf{set}(E), \mathit{do}(\vec{a}, S_0))$, $M \models A_u(a_u, \mathit{do}(\vec{a}\vec{a'}, S_0))$, $\vec{a'}a_u\vec{d} \in \mathcal{CR}_M(\delta^*, \mathit{do}(\vec{a}, S_0))$, and for all ground action sequences $\vec{d'}$, $\vec{a'}a_u\vec{d'} \notin \mathcal{CR}_M(\mathbf{set}(E), \mathit{do}(\vec{a}, S_0))$. Then we have two cases. If $M \models \exists s'. \mathit{Do}(\delta, \mathit{do}(\vec{a}, S_0), s') \wedge \mathit{do}(\vec{a}\vec{a'}, S_0) < s'$, then $M \models \neg \mathit{Controllable}(E^1, \delta, \mathit{do}(\vec{a}, S_0))$, contradiction.
If on the other hand $M \models \neg\exists s'. \mathit{Do}(\delta, \mathit{do}(\vec{a}, S_0), s') \wedge \mathit{do}(\vec{a}\vec{a'}, S_0) < s'$, then there exists ground action sequences $\vec{b}$ and $\vec{b'}$ such that $\vec{a'} = \vec{b}\vec{b'}$, $M \models \mathit{Do}(\delta, \mathit{do}(\vec{a}, S_0), \mathit{do}(\vec{a}\vec{b}, S_0))$, and $M \models \exists s'. \mathit{Do}(\delta^*, \mathit{do}(\vec{a}, S_0), s') \wedge \mathit{do}(\vec{a}\vec{b}\vec{b'}, S_0) < s'$. This implies $M \models \neg \mathit{Controllable}(E^{\vec{b}}, \delta^*, \mathit{do}(\vec{a}\vec{b}, S_0))$, contradiction.
Now suppose that we also have that $M \models \mathit{Controllable}(\mathbf{set}(\{\epsilon\}), \delta^*, \mathit{do}(\vec{a}, S_0))$. It follows that $M \models \neg\exists a_u. \mathit{Poss}(a_u, \mathit{do}(\vec{a}, S_0)) \wedge A_u(a_u, \mathit{do}(\vec{a}, S_0))$. Thus we also have that $M \models \mathit{Controllable}(\mathbf{set}(E \cup \{\epsilon\}), \delta^*, \mathit{do}(\vec{a}, S_0))$. $\square$

Lemma A.10 *For any sequence of ground low-level actions $\vec{a}$, and any set of ground low-level action sequences E_l we have that if $M_l \models \mathit{Controllable}(\mathbf{set}(E_l), \textsc{monit}, \mathit{do}(\vec{a}, S_0))$ then $M_l \models \mathit{Controllable}(\mathbf{set}(E^1_l), \textsc{oneMonit}, \mathit{do}(\vec{a}, S_0))$ where $E^1_l = \{\vec{b} \mid \vec{b}\vec{c} \in E_l$ and $M_l \models \mathit{Do}(\textsc{oneMonit}, \mathit{do}(\vec{a}, S_0), \mathit{do}(\vec{a}\vec{b}, S_0))$, for some $\vec{c}\}$.*

Proof By contradiction. Assume the antecedent and the negation of the consequent. The latter means that there is a sequence of actions $\vec{b} = \vec{b_1}b_u\vec{b_2}$ such that $A_l^u(b_u, do(\vec{a}\vec{b_1}, S_0))$ and $\vec{b} \in \mathcal{CR}_{M_l}(\textsc{oneMonit}, do(\vec{a}, S_0))$, and there is another sequence of actions $\vec{c} = \vec{b_1}c\vec{c_2}$ such that $\vec{c} \in \mathcal{CR}_{M_l}(\mathbf{set}(E_l^1), do(\vec{a}, S_0))$ and $\vec{b} \notin \mathcal{CR}_{M_l}(\mathbf{set}(E_l^1), do(\vec{a}, S_0))$ ($\vec{b_1}$ is the shared prefix).

By definition we have that E_l^1 is the prefix of E_l. This means there exists sequence of actions $\vec{d}$ and $\vec{e}$ such that $\vec{b}\vec{d} \in \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ and $\vec{c}\vec{e} \in \mathcal{CR}_{M_l}(\mathbf{set}(E_l), do(\vec{a}, S_0))$ and $\vec{b}\vec{d} \notin \mathcal{CR}_{M_l}(\mathbf{set}(E_l), do(\vec{a}, S_0))$. By antecedent we know that $M_l \models \mathit{Controllable}(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$, a contradiction. □

Lemma A.11 *For any sequence of ground high-level actions $\vec{\alpha}$, and any set of ground high-level actions E_h we have that if $M_h \models \mathcal{C}$ and $M_h \models \mathit{Controllable}(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$ then $M_h \models \mathit{Controllable}(\mathbf{set}(E_h^1), \textsc{anyOne}, do(\vec{\alpha}, S_0))$ where $E_h^1 = \{\beta \mid \beta\vec{\gamma} \in E_h\}$.*

Proof By contradiction. Assume the antecedent and the negation of the consequent. The latter means that there is an uncontrollable action β_u such that $\beta_u \in \mathcal{CR}_{M_h}(\textsc{anyOne}, do(\vec{\alpha}, S_0))$, but $\beta_u \notin \mathcal{CR}_{M_h}(\mathbf{set}(E_h^1), do(\vec{\alpha}, S_0))$.

By definition we have that E_h^1 is the prefix of E_h. This means there exists sequence of actions $\vec{\gamma}$ such that $\beta_u\vec{\gamma} \in \mathcal{CR}_{M_h}(\textsc{any}, do(\vec{\alpha}, S_0))$ and $\beta_u\vec{\gamma} \notin \mathcal{CR}_{M_h}(\mathbf{set}(E_h), do(\vec{\alpha}, S_0))$ By antecedent we know that $M_h \models \mathit{Controllable}(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$, a contradiction. □

Theorem 6.2 If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 (part(a) $\supset$) hold, then

for any set of ground high-level action sequences of bounded length E_h such that $M_h \models \mathit{Controllable}(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$, there exists a set of ground low-level action sequences E_l such that $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ and $M_l \models \mathit{Controllable}(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$ and $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0)) = E_h$.

Proof By induction on the length of the longest action sequence in E_h. Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent and that Assumptions 5.1 and 6.1 hold.
Base case: If $E_h = \emptyset$, then we can choose $E_l = \emptyset$ as $\emptyset$ is always controllable. If $E_h = \{\epsilon\}$, then by Assumption 6.1, we can choose $E_l = \{\epsilon\}$ and the result follows.
Inductive step: Assume that the thesis holds for all E_h such that $max_{\vec{\gamma} \in E_h}|\vec{\gamma}| \leq N$. Take an arbitrary set of ground high-level action sequences E_h such that $max_{\vec{\gamma} \in E_h}|\vec{\gamma}| \leq N + 1$, and assume that $M_h \models \mathit{Executable}(do(\vec{\alpha}, S_0))$, $M_l \models \mathit{Do}(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$, and $M_h \models \mathit{Controllable}(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$. Let $E_h^1 = \{\gamma \mid \gamma\vec{\beta} \in E_h \text{ for some } \vec{\beta}\}$ and let $E_h^{\gamma} = \{\vec{\beta} \mid \gamma\vec{\beta} \in E_h\}$. Clearly for all $\gamma \in E_h^1$, $M_h \models \mathit{Controllable}(\mathbf{set}(E_h^{\gamma}), \textsc{any}, do(\vec{\alpha}\gamma, S_0))$ and $max_{\vec{\beta} \in E_h^{\gamma}}|\vec{\beta}| \leq N$. Take a $\gamma \in E_h^1$ and an arbitrary ground low-level action sequence $\vec{c}$ such that $M_l \models \mathit{Do}(m(\gamma), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0))$. By the induction hypothesis there exists a set of ground low-level action sequences $E_l^{\gamma,\vec{c}}$ such that $M_l \models \mathit{Controllable}(\mathbf{set}(E_l^{\gamma,\vec{c}}), \textsc{monit}, do(\vec{a}\vec{c}, S_0))$ and $m_{M_l}^{-1}(E_l^{\gamma,\vec{c}}, do(\vec{a}\vec{c}, S_0)) = E_h^{\gamma}$. We can also show that $M_h \models \mathit{Controllable}(\mathbf{set}(E_h^1), \textsc{anyOne}, do(\vec{\alpha}, S_0))$ by Lemma A.11. By Assumption 6.1, there exists a set of ground low-level action sequences E_l^1 such that $M_l \models \mathit{Controllable}(\mathbf{set}(E_l^1), \textsc{oneMonit}, (\vec{a}, S_0))$ and $m_{M_l}^{-1}(E_l^1, (\vec{a}, S_0)) = E_h^1$. If $\epsilon \in E_h$, we have that $M_l \models \mathit{Controllable}(\mathbf{set}(\{\epsilon\}), \textsc{monit}, do(\vec{a}, S_0))$. Let

$$\begin{aligned} E_l = \{\vec{c}\vec{b} \mid\ & \vec{c} \in E_l^1 \text{ and} \\ & M_l \models \mathit{Do}(m(\gamma), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0)) \text{ and} \\ & \vec{b} \in E_l^{\gamma,\vec{c}} \text{ for some } \gamma\} \cup Q \end{aligned}$$

where $Q = \{\epsilon\}$ if $\epsilon \in E_h$ and $Q = \emptyset$ otherwise. It follows by Lemma A.9 that $M_l \models \mathit{Controllable}(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$.

Now let's show that $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0)) = E_h$. First let's show that $E_h \subseteq m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))$. Take an arbitrary $\vec{\beta}' \in E_h$. If $\vec{\beta}' = \epsilon$, then $\epsilon \in E_l$ and $m_{M_l}^{-1}(\epsilon, (\vec{a}, S_0)) = \epsilon$. Otherwise $\vec{\beta}' = \gamma\vec{\beta}$. We have that

$\gamma \in E_h^1$ and $m_{M_l}^{-1}(E_l^1, do(\vec{a}, S_0)) = E_h^1$, so there exists a ground low-level action sequence $\vec{c} \in E_l^1$ such that $m_{M_l}^{-1}(\vec{c}, do(\vec{a}, S_0)) = \gamma$. We also have that $m_{M_l}^{-1}(E_l^{\gamma,\vec{c}}, do(\vec{a}\vec{c}, S_0)) = E_h^{\gamma}$, so there exists $\vec{b} \in E_l^{\gamma,\vec{c}}$ such that $m_{M_l}^{-1}(\vec{b}, do(\vec{a}\vec{c}, S_0)) = \vec{\beta}$. We have that $\vec{c}\vec{b} \in E_l$, therefore $\gamma\vec{\beta} \in m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))$.

Finally, let's show that $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0)) \subseteq E_h$. Take an arbitrary $\vec{e} \in E_l$. If $\vec{e} = \epsilon$, then $m_{M_l}^{-1}(\epsilon, (\vec{a}, S_0)) = \epsilon$ and $\epsilon \in E_h$. Otherwise, $\vec{e} = \vec{c}\vec{b}$ where $\vec{c} \neq \epsilon$, $\vec{c} \in E_l^1$, and $\vec{b} \in E_l^{\gamma,\vec{c}}$ for some γ such that $M_l \models Do(m(\gamma), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0))$. Then $m_{M_l}^{-1}(\vec{c}, do(\vec{a}, S_0)) \in E_h^1$, since $m_{M_l}^{-1}(E_l^1, do(\vec{a}, S_0)) = E_h^1$. Moreover we have that $m_{M_l}^{-1}(\vec{b}, (\vec{a}\vec{c}, S_0)) \in E_h^{\gamma}$, since $m_{M_l}^{-1}(E_l^{\gamma,\vec{c}}, do(\vec{a}\vec{c}, S_0)) = E_h^{\gamma}$. Thus $m_{M_l}^{-1}(\vec{c}\vec{b}, do(\vec{a}, S_0)) \in E_h$.

□

Theorem 6.3 If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 (part(a) $\subset$ and part (b)) hold, then

for any set of ground low-level action sequences E_l such that $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))$ has bounded length,

if $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$,
then $M_h \models Controllable(\mathbf{set}(m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))), \textsc{any}, do(\vec{\alpha}, S_0))$.

Proof By induction on the length of the longest action sequence in $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))$. Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent. Also suppose that Assumptions 5.1 and 6.1 hold.
Base case: If $E_l = \emptyset$, then $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0)) = \emptyset$ and $\emptyset$ is always controllable. If $E_l = \{\epsilon\}$, then $m_{M_l}^{-1}(E_l, do(\vec{a}, S_0)) = \{\epsilon\}$, and by Assumption 6.1 the result holds.
Inductive step: Assume that the thesis holds for all E_l such that $max_{\vec{\gamma} \in m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))}|\vec{\gamma}| \leq N$. Take an arbitrary set of ground high-level action sequences E_l such that $max_{\vec{\gamma} \in m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))}|\vec{\gamma}| \leq N + 1$, and assume that $M_h \models Executable(do(\vec{\alpha}, S_0))$, $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$, $E_l \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$, and $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$. Let $E_l^1 = \{\vec{b} \mid \vec{b}\vec{c} \in E_l$ and $M_l \models Do(\textsc{oneMonit}, do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$, for some $\vec{c}\}$ and let $E_l^{\vec{b}} = \{\vec{c} \mid \vec{b}\vec{c} \in E_l$ and $M_l \models Do(\textsc{oneMonit}, do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))\}$. Clearly for all $\vec{b} \in E_l^1$, $E_l^{\vec{b}} \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}\vec{b}, S_0))$, $M_l \models Controllable(\mathbf{set}(E_l^{\vec{b}}), \textsc{monit}, do(\vec{a}\vec{b}, S_0))$, and $max_{\vec{\beta} \in m_{M_l}^{-1}(E_l^{\vec{b}}, do(\vec{a}\vec{b}, S_0))}|\vec{\beta}| \leq N$. Thus by the induction hypothesis

$$M_h \models Controllable(\mathbf{set}(m_{M_l}^{-1}(E_l^{\vec{b}}, do(\vec{a}\vec{b}, S_0)), \textsc{any}, do(m_{M_l}^{-1}(do(\vec{a}\vec{b}, S_0))).$$

We can also show that $M_l \models Controllable(\mathbf{set}(E_l^1), \textsc{oneMonit}, do(\vec{a}, S_0))$ by Lemma A.10. By Assumption 6.1, we must then also have that $M_h \models Controllable(\mathbf{set}(m_{M_l}^{-1}(E_l^1, do(\vec{a}, S_0))), \textsc{anyOne}, do(\vec{\alpha}, S_0))$. If $\epsilon \in E_l$, we have that $M_h \models Controllable(\mathbf{set}(\{\epsilon\}), \textsc{any}, do(\vec{\alpha}, S_0))$. Let

$$E_h = \{m_{M_l}^{-1}(\vec{b}, do(\vec{a}, S_0))\, m_{M_l}^{-1}(\vec{c}, do(\vec{a}\vec{b}, S_0)) \mid \vec{b} \in E_l^1 \text{ and } \vec{c} \in E_l^{\vec{b}}\} \cup Q$$

where $Q = \{\epsilon\}$ if $\epsilon \in E_l$ and $Q = \emptyset$ otherwise. It follows by Lemma A.9 that $M_h \models Controllable(\mathbf{set}(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$. Clearly $E_h = m_{M_l}^{-1}(E_l, do(\vec{a}, S_0))$ and the result follows. □

Here are some results about **setp**:

Lemma A.12 *For any BAT $\mathcal{D}$, $\mathcal{D} \cup \mathcal{C} \models \forall s, P.SituationDetermined(\mathbf{setp}(P), s)$.*

Proof (Sketch) By induction on the number of transitions, i.e., offline executions from $(\mathbf{setp}(P), s)$. Based on definition of *Trans* for **setp**, whenever an action a is performed in any situation s, there is a unique remaining program in the resulting situation. This clearly extends to any configurations reachable through $Trans^*$ from $(\mathbf{setp}(P), s)$. □

Lemma A.13 *For any BAT $\mathcal{D}$,*

$$\begin{aligned}\mathcal{D} \cup \mathcal{C} \models\ & \mathit{Trans}^*(\mathbf{setp}(P), s, \delta, s') \equiv \\ & \exists \delta', \delta''.\delta' \in P \wedge \mathit{Trans}^*(\delta', s, \delta'', s') \wedge \\ & \delta = \mathbf{setp}(\{\delta'' \mid \exists \delta'.\delta' \in P \wedge \mathit{Trans}^*(\delta', s, \delta'', s')\})\end{aligned}$$

Proof (Sketch) By induction on number of transitions. □

We introduce the abbreviation $\mathit{sub}_m(\delta)$ which replaces any primitive action α by $m(\alpha)$ and any fluent $F(\vec{x})$ by $m(F)[\vec{x}]$:

$$\begin{aligned}&\mathit{sub}_m(\mathit{nil}) \doteq m(\mathit{nil}) = \mathit{nil}\\ &\mathit{sub}_m(\alpha) \doteq \mathbf{atomic}(m(\alpha))\\ &\mathit{sub}_m(\varphi?) \doteq m(\phi)?\\ &\mathit{sub}_m(\delta_1;\delta_2) \doteq \mathit{sub}_m(\delta_1);\mathit{sub}_m(\delta_2)\\ &\mathit{sub}_m(\delta_1 \mid \delta_2) \doteq \mathit{sub}_m(\delta_1) \mid \mathit{sub}_m(\delta_2)\\ &\mathit{sub}_m(\pi x.\delta) \doteq \pi x.\mathit{sub}_m(\delta)\\ &\mathit{sub}_m(\delta^*) \doteq \mathit{sub}_m(\delta)^*\\ &\mathit{sub}_m(\delta_1 \| \delta_2) \doteq \mathit{sub}_m(\delta_1) \| \mathit{sub}_m(\delta_2)\\ &\mathit{sub}_m(\delta_1 \ \&\ \delta_2) \doteq \mathit{sub}_m(\delta_1)\ \&\ \mathit{sub}_m(\delta_2)\end{aligned}$$

It is easy to show by induction on the structure of δ that $\mathit{sub}_m(\delta) = \delta[A(\vec{x})/\mathbf{atomic}(m(A(\vec{x})))$ for all $A \in \mathcal{A}$, and $F(\vec{x})/m(F(\vec{x}))$ for all $F \in \mathcal{F}]\}]$.

Lemma A.14 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent and that $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds. Then, for any high-level program δ_h, we have that*

$$M_l \models \mathit{Final}(\mathit{sub}_m(\delta_h), \mathit{do}(\vec{a}, S_0)) \text{ iff } M_h \models \mathit{Final}(\delta_h, \mathit{do}(\vec{\alpha}, S_0))$$

Proof By induction on the structure of δ_h.
Base cases: nil, α, $\phi?$.
Case $\delta_h = \mathit{nil}$.
We have $\mathit{sub}_m(\mathit{nil}) = m(\mathit{nil}) = \mathit{nil}$. By the axioms for *Final*, we have that $M_h \models \mathit{Final}(\mathit{nil}, \mathit{do}(\vec{\alpha}, S_0)) \equiv \mathtt{True}$, and also, $M_l \models \mathit{Final}(\mathit{nil}, \mathit{do}(\vec{a}, S_0)) \equiv \mathtt{True}$. Thus the result holds.
Case $\delta_h = \alpha$.
By the axioms for *Final*, we have that $M_h \models \mathit{Final}(\alpha, \mathit{do}(\vec{\alpha}, S_0)) \equiv \mathtt{False}$. By Assumption 5.1 part (c) we know that $m(\alpha) \neq \mathit{nil}$ and $m(\alpha) \neq \varphi?$. From this and by the axioms for *Final*, we have that $M_l \models \mathit{Final}(\mathbf{atomic}(m(\alpha)), \mathit{do}(\vec{a}, S_0)) \equiv \mathtt{False}$. Thus the result holds.
Case $\delta_h = \varphi?$.
By the axioms for *Final*, we have that $M_h \models \mathit{Final}(\varphi?, \mathit{do}(\vec{\alpha}, S_0)) \equiv \varphi[\mathit{do}(\vec{\alpha}, S_0)]$ and that $M_l \models \mathit{Final}(m(\varphi)?, \mathit{do}(\vec{a}, S_0)) \equiv m(\varphi)[\mathit{do}(\vec{a}, S_0)]$. Since $M_h \sim_m M_l$, by Lemma 5.1, we have $M_h \models \varphi[\mathit{do}(\vec{\alpha}, S_0)]$ iff $M_l \models m(\varphi)[\mathit{do}(\vec{a}, S_0)]$. Thus the result follows.

Induction step:
Case $\delta_h = \delta_1; \delta_2$.

Assume the results hold for δ_1 and δ_2 (IH).

$M_l \models \mathit{Final}(sub_m(\delta_1;\delta_2), do(\vec{a}, S_0))$
iff
$M_l \models \mathit{Final}((sub_m(\delta_1); sub_m(\delta_2)), do(\vec{a}, S_0))$ (By definition of sub_m)
iff
$M_l \models \mathit{Final}(sub_m(\delta_1), do(\vec{a}, S_0))) \wedge \mathit{Final}(sub_m(\delta_1), do(\vec{a}, S_0))$ (By Final axioms for ;)
iff
$M_h \models \mathit{Final}(\delta_1, do(\vec{\alpha}, S_0)) \wedge \mathit{Final}(\delta_2, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models \mathit{Final}(\delta_1;\delta_2, do(\vec{\alpha}, S_0))$ (By Final axioms for ;)

Case $\delta_h = \delta_1 \mid \delta_2$.
Assume the results hold for δ_1 and δ_2 (IH).

$M_l \models \mathit{Final}(sub_m(\delta_1 \mid \delta_2), do(\vec{a}, S_0))$
iff
$M_l \models \mathit{Final}((sub_m(\delta_1) \mid sub_m(\delta_2)), do(\vec{a}, S_0))$ (By definition of sub_m)
iff
$M_l \models \mathit{Final}(sub_m(\delta_1), do(\vec{a}, S_0))) \vee \mathit{Final}(sub_m(\delta_1), do(\vec{a}, S_0))$ (By Final axioms for $|$)
iff
$M_h \models \mathit{Final}(\delta_1, do(\vec{\alpha}, S_0)) \vee \mathit{Final}(\delta_2, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models \mathit{Final}(\delta_1 \mid \delta_2, do(\vec{\alpha}, S_0))$ (By Final axioms for $|$)

Case $\delta_h = \pi x.\delta$.
Assume the results hold for δ (IH).

$M_l \models \mathit{Final}(sub_m(\pi x.\delta), do(\vec{a}, S_0))$
iff
$M_l \models \mathit{Final}(\pi x.sub_m(\delta), do(\vec{a}, S_0))$
iff
$M_l \models \exists x.\mathit{Final}(sub_m(\delta), do(\vec{a}, S_0)))$ (By Final axioms for πx)
iff
$M_h \models \exists x.\mathit{Final}(\delta, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models \mathit{Final}(\pi x.\delta, do(\vec{\alpha}, S_0))$ (By Final axioms for πx)

Case $\delta_h = \delta^*$.
Assume the results hold for δ (IH).

$M_l \models \mathit{Final}(sub_m(\delta^*), do(\vec{a}, S_0))$
iff
$M_l \models \mathit{Final}(sub_m(\delta)^*, do(\vec{a}, S_0))$
iff
$M_l \models \texttt{True}$ (By Final axioms for $*$)
iff
$M_h \models \mathit{Final}(\delta, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models \mathit{Final}(\delta^*, do(\vec{\alpha}, S_0))$ (By Final axioms for $*$)

Case $\delta_h = \delta_1 \| \delta_2$.

Assume the results hold for δ_1 and δ_2 (IH).

$M_l \models Final(sub_m(\delta_1 \| \delta_2), do(\vec{a}, S_0))$
iff
$M_l \models Final((sub_m(\delta_1) \| sub_m(\delta_2)), do(\vec{a}, S_0))$ (By definition of sub_m)
iff
$M_l \models Final(sub_m(\delta_1), do(\vec{a}, S_0))) \wedge Final(sub_m(\delta_1), do(\vec{a}, S_0))$ (By Final axioms for $\|$)
iff
$M_h \models Final(\delta_1, do(\vec{\alpha}, S_0)) \wedge Final(\delta_2, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models Final(\delta_1 \| \delta_2, do(\vec{\alpha}, S_0))$ (By Final axioms for $\|$)

Case $\delta_h = \delta_1 \ \& \ \delta_2$.
Assume the results hold for δ_1 and δ_2 (IH).

$M_l \models Final(sub_m(\delta_1 \ \& \ \delta_2), do(\vec{a}, S_0))$
iff
$M_l \models Final((sub_m(\delta_1) \ \& \ sub_m(\delta_2)), do(\vec{a}, S_0))$ (By definition of sub_m)
iff
$M_l \models Final(sub_m(\delta_1), do(\vec{a}, S_0))) \wedge Final(sub_m(\delta_1), do(\vec{a}, S_0))$ (By Final axioms for &)
iff
$M_h \models Final(\delta_1, do(\vec{\alpha}, S_0)) \wedge Final(\delta_2, do(\vec{\alpha}, S_0))$ (By IH)
iff
$M_h \models Final(\delta_1 \ \& \ \delta_2, do(\vec{\alpha}, S_0))$ (By Final axioms for &)

□

Corollary A.15 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds. Then, for any high-level program δ_h such that $M_h \models SituationDetermined(\delta_h, do(\vec{\alpha}, S_0))$, we have that*

$$M_l \models Final(m_p(\delta_h), do(\vec{a}, S_0)) \text{ iff } M_h \models Final(\delta_h, do(\vec{\alpha}, S_0))$$

Proof

$M_l \models Final(\mathbf{setp}((sub_m(\delta_h))), do(\vec{a}, S_0))$
iff
$M_l \models Final(sub_m(\delta_h), do(\vec{a}, S_0))$ By axioms for $\mathbf{setp}()$
iff
$M_l \models Final(\delta_h, do(\vec{a}, S_0))$ By Lemma A.14

□

Here are some results about m_p:

Lemma A.16 *For any ConGolog programs δ, δ_1 and δ_2:*

1. $\mathcal{C} \models Do(\delta_1; \delta_2, s, s') \equiv \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$,
2. $\mathcal{C} \models Do(\delta_1 \mid \delta_2, s, s') \equiv Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$,
3. $\mathcal{C} \models Do(\delta_1 \ \& \ \delta_2, s, s') \equiv Do(\delta_1, s, s') \wedge Do(\delta_2, s, s')$,

4. $\mathcal{C} \models Do(\pi x.\delta, s, s') \equiv \exists x.Do(\delta, s, s')$,

5. *for every model M of $\mathcal{C}$ $M \models Do(\delta^*, s, s')$ if and only if there exists $s_1, \ldots, s_n$ such that $s = s_1$ and $s' = s_n$ and for all $i = 1, \ldots, n-1$, $M \models Do(\delta, s_i, s_{i+1})$.*

Proof We provide a detailed proof for (1): Take an arbitrary model M of $\mathcal{C}$. By Lemma B.1 of [33] and the definition of Do, $M \models Do(\delta_a; \delta_b, s, s')$ holds if and only if there exists $\delta_1, s_1, \ldots, \delta_n, s_n$ such that $\delta_1 = \delta_a; \delta_b$, $s_1 = s$, and $s_n = s'$ and for all $i = 1, \ldots, n-1$, $M \models Trans(\delta_i, s_i, \delta_{i+1}, s_{i+1})$ and $M \models Final(\delta_n, s_n)$. By the $\mathcal{C}$ axioms, this holds if and only if there exists $\delta_1, s_1, \ldots, \delta_n, s_n$ such that $\delta_1 = \delta_a$, $s_1 = s$, and $s_n = s'$ and for all $i = 1, \ldots, k-1$, $M \models Trans(\delta_i; \delta_b, s_i, \delta_{i+1}; \delta_b, s_{i+1})$, $M \models Final(\delta_k, s_k)$, $M \models Trans(\delta_k; \delta_b, s_k, \delta_{k+1}, s_{k+1})$, and for all $j = k+1, \ldots, n-1$, $M \models Trans(\delta_j, s_j, \delta_{j+1}, s_{j+1})$, and $M \models Final(\delta_n, s_n)$. This holds if and only if there exists $\delta_1, s_1, \ldots, \delta_k, s_k$ such that $\delta_1 = \delta_a$, and $s = s_1$, and for all $i = 1, \ldots, k-1$, $M \models Trans(\delta_i, s_i, \delta_{i+1}; \delta_b, s_{i+1})$, and $M \models Final(\delta_k, s_k)$, and there exists $\delta_{k+1}, s_{k+1}, \ldots, \delta_n, s_n$ such that $\delta_{k+1} = \delta_b$, $s_n = s'$ and for all $j = k, \ldots, n-1$, $M \models Trans(\delta_j, s_j, \delta_{j+1}, s_{j+1})$, and $M \models Final(\delta_n, s_n)$. By Lemma B.1 of [33] and the definition of Do, this holds if and only if $M \models \exists s''.Do(\delta_a, s, s'') \wedge Do(\delta_b, s'', s')$.

The other cases can be proven in a similar way. □

Let Golog $^+$ stand for the Golog language extended with the intersection construct &.

We can show that:

Lemma A.17 *If $M_h \sim_m M_l$, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds, then for any high-level Golog $^+$ program δ_h such that $M_h \models SituationDetermined(\delta_h, do(\vec{\alpha}, S_0))$, any ground high-level action sequence $\vec{\beta}$, and any ground low-level action sequence $\vec{b}$, if*

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \text{ and}$$
$$M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

then

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

Proof By induction of the structure of δ_h.
Base cases:
Case $\delta_h = \beta$, where β is a primitive action:
Then $m_p(\delta_h) = \textbf{setp}(\{\textbf{atomic}(m(\beta))\})$. We have that $M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$, and thus $M_h \models \vec{\beta} = \beta$. Then we have that $M_l \models Do(m(\beta), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$. It follows by properties of **setp** and the $\mathcal{C}$ axioms that $M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$.
Case $\delta_h = \phi?$:
Then $m_p(\delta_h) = \textbf{setp}(\{m(\phi?)\})$. We have that $M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$. It follows by the $\mathcal{C}$ axioms that $M_h \models \phi[do(\vec{\alpha}, S_0)] \wedge \vec{\beta} = \epsilon$. We have that $M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ and thus $M_l \models \vec{b} = \epsilon$. Since $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, it follows by Theorem 5.2 that $M_l \models m(\phi)[do(\vec{a}, S_0)]$. It then follows by the properties of **setp** and the $\mathcal{C}$ axioms that $M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$.
Case $\delta_h = nil$:
Then $m_p(\delta_h) = \textbf{setp}(\{nil\})$. We have that $M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$, and thus $M_h \models \vec{\beta} = \epsilon$. We have that $M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ and thus $M_l \models \vec{b} = \epsilon$. It follows that the claim holds by the properties of **setp** and the $\mathcal{C}$ axioms.

Inductive step:
Assume that the claim holds for all the proper subprograms of δ_h (IH). Let's show it must then hold for δ_h.
Case $\delta_h = \delta_1; \delta_2$:

Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1); sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$\begin{array}{l} M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \text{ and} \\ M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

It follows by Lemma A.16 that there exist $\vec{\gamma}$ and $\vec{\gamma}'$ such that $\vec{\beta} = \vec{\gamma}\vec{\gamma}'$ and

$$\begin{array}{l} M_h \models Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\gamma}, S_0)) \wedge \\ \qquad Do(\delta_2, do(\vec{\alpha}\vec{\gamma}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \end{array}$$

It also follows by Lemma A.16 that there exist $\vec{c}$ and $\vec{c}'$ such that $\vec{b} = \vec{c}\vec{c}'$ and

$$\begin{array}{l} M_l \models Do(m(\vec{\gamma}), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0)) \wedge \\ \qquad Do(m(\vec{\gamma}'), do(\vec{a}\vec{c}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

Then by the IH, we have that

$$\begin{array}{l} M_l \models Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0)) \wedge \\ \qquad Do(m_p(\delta_2), do(\vec{a}\vec{c}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

Case $\delta_h = \delta_1 \mid \delta_2$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1) \mid sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$\begin{array}{l} M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \text{ and} \\ M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

It follows by Lemma A.16 that

$$\begin{array}{l} M_h \models Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \vee \\ \qquad Do(\delta_2, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \end{array}$$

Then by the IH, we have that

$$\begin{array}{l} M_l \models Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \vee \\ \qquad Do(m_p(\delta_2), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

Case $\delta_h = \pi x.\delta_1$:
the proof is similar to that for $\delta_h = \delta_1 \mid \delta_2$.
Case $\delta_h = \delta_1^*$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1)^*)\})$. Assume the antecedent. Then we have that

$$\begin{array}{l} M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \text{ and} \\ M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{array}$$

It follows by Lemma A.16 that there exist $\vec{\gamma_1}, \ldots, \vec{\gamma_n}$ such that $\vec{\beta} = \vec{\gamma_1} \ldots \vec{\gamma_n}'$ and

$$M_h \models \bigwedge_{i:1..n} Do(\delta_1, do(\vec{\alpha}\vec{\gamma_1} \ldots \vec{\gamma_{i-1}}, S_0), do(\vec{\alpha}\vec{\gamma_1} \ldots \vec{\gamma_i}, S_0))$$

It also follows by Lemma A.16 that there exist $\vec{c_1}, \ldots, \vec{c_n}$ such that $\vec{b} = \vec{c_1} \ldots \vec{c_n}$ and

$$M_l \models \bigwedge_{i:1..n} Do(m(\vec{\gamma_i}), do(\vec{a}\vec{c_1} \ldots \vec{c_{i-1}}, S_0), do(\vec{a}\vec{c_1} \ldots \vec{c_i}, S_0))$$

Then by the IH, we have that

$$M_l \models \bigwedge_{i:1..n} Do(m_p(\delta_1), do(\vec{a}\vec{c_1} \ldots \vec{c_{i-1}}, S_0), do(\vec{a}\vec{c_1} \ldots \vec{c_i}, S_0))$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

Case $\delta_h = \delta_1 \ \& \ \delta_2$:
Then $m_p(\delta_h) = \textbf{setp}(\{sub_m(\delta_1) \ \& \ sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \text{ and}$$
$$M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

It follows by Lemma A.16 that

$$M_h \models Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \wedge$$
$$Do(\delta_2, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

Then by the IH, we have that

$$M_l \models Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge$$
$$Do(m_p(\delta_2), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

□

We can also show that:

Lemma A.18 *If $M_h \sim_m M_l$, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds, then for any high-level Golog$^+$ program δ_h such that $M_h \models SituationDetermined(\delta_h, do(\vec{\alpha}, S_0))$, any ground high-level action sequence $\vec{\beta}$, and any ground low-level action sequence $\vec{b}$, if*

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge$$
$$Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

then

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

Proof By induction of the structure of δ_h.
Base cases:
Case $\delta_h = \beta$, where β is a primitive action:
Then $m_p(\delta_h) = \textbf{setp}(\{\textbf{atomic}(m(\beta))\})$. We have that $M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ and $M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$, and thus $M_l \models \vec{\beta} = \beta$. Since $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, it follows by Theorem 5.2 that $M_h \models Poss(\beta, do(\vec{\alpha}, S_0))$, and the claim then follows by the $\mathcal{C}$ axioms.
Case $\delta_h = \phi?$:
Then $m_p(\delta_h) = \textbf{setp}(\{m(\phi?)\})$. We have that $M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ and thus that $M_l \models$

$m(\phi)[do(\vec{a}, S_0)] \wedge \vec{b} = \epsilon$. Since $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, it follows by Theorem 5.2 that $M_h \models \phi[do(\vec{\alpha}, S_0)]$. We also have that $M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$, and thus $M_l \models \vec{\beta} = \epsilon$. Then the claim follows by the $\mathcal{C}$ axioms.
Case $\delta_h = nil$:
Then $m_p(\delta_h) = \mathbf{setp}(\{nil\})$. We have that $M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ and thus $M_l \models \vec{b} = \epsilon$. We also have that $M_l \models Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$, and thus $M_l \models \vec{\beta} = \epsilon$. It follows that the claim holds by the $\mathcal{C}$ axioms.

Inductive step:
Assume that the claim holds for all the proper subprograms of δ_h (IH). Let's show it must then hold for δ_h.
Case $\delta_h = \delta_1; \delta_2$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1); sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$\begin{aligned} M_l \models\ & Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ & Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{aligned}$$

It follows by properties of **setp** and Lemma A.16 that there exist $\vec{c}$ and $\vec{c}'$ such that $\vec{b} = \vec{c}\vec{c}'$ and there exist $\vec{\gamma}$ and $\vec{\gamma}'$ such that $\vec{\beta} = \vec{\gamma}\vec{\gamma}'$ and

$$\begin{aligned} M_l \models\ & Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0)) \wedge \\ & Do(m_p(\delta_2), do(\vec{a}\vec{c}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ & Do(m(\vec{\gamma}), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0)) \wedge \\ & Do(m(\vec{\gamma}'), do(\vec{a}\vec{c}, S_0), do(\vec{a}\vec{b}, S_0)) \end{aligned}$$

Then by the IH, we have that

$$\begin{aligned} M_h \models\ & Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\gamma}, S_0)) \wedge \\ & Do(\delta_2, do(\vec{\alpha}\vec{\gamma}', S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \end{aligned}$$

By Lemma A.16, it follows that

$$M_h \models Do(\delta_h, do(\vec{a}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

Case $\delta_h = \delta_1 \mid \delta_2$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1) \mid sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$\begin{aligned} M_l \models\ & Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ & Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{aligned}$$

It follows by properties of **setp** and Lemma A.16 that

$$\begin{aligned} M_l \models\ & Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \vee \\ & Do(m_p(\delta_2), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \end{aligned}$$

Then by the IH, we have that

$$\begin{aligned} M_h \models\ & Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \vee \\ & Do(\delta_2, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \end{aligned}$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

Case $\delta_h = \pi x.\delta_1$:
the proof is similar to that for $\delta_h = \delta_1 \mid \delta_2$.

Case $\delta_h = \delta_1^*$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1)^*\})$. Assume the antecedent. Then we have that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

It follows by properties of **setp** and Lemma A.16 that there exist $\vec{c_1}, \ldots, \vec{c_n}$ and $\vec{\gamma_1}, \ldots, \vec{\gamma_n}$ such that $\vec{b} = \vec{c_1} \ldots \vec{c_n}$ and $\vec{\beta} = \vec{\gamma_1} \ldots \vec{\gamma_n}$ and

$$M_l \models \bigwedge_{i:1..n} Do(m_p(\delta_1), do(\vec{a}\vec{c_1} \ldots \vec{c_{i-1}}, S_0), do(\vec{a}\vec{c_1} \ldots \vec{c_i}, S_0)) \wedge \\ \bigwedge_{i:1..n} Do(m(\vec{\gamma_i}), do(\vec{a}\vec{c_1} \ldots \vec{c_{i-1}}, S_0), do(\vec{a}\vec{c_1} \ldots \vec{c_i}, S_0))$$

Then by the IH, we have that

$$M_h \models \bigwedge_{i:1..n} Do(\delta_1, do(\vec{\alpha}\vec{\gamma_1} \ldots \vec{\gamma_{i-1}}, S_0), do(\vec{\alpha}\vec{\gamma_1} \ldots \vec{\gamma_i}, S_0))$$

By Lemma A.16, it follows that

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

Case $\delta_h = \delta_1 \;\&\; \delta_2$:
Then $m_p(\delta_h) = \mathbf{setp}(\{sub_m(\delta_1) \;\&\; sub_m(\delta_2))\})$. Assume the antecedent. Then we have that

$$M_l \models Do(m_p(\delta_h), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ Do(m(\vec{\beta}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

It follows by properties of **setp** and Lemma A.16 that

$$M_l \models Do(m_p(\delta_1), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0)) \wedge \\ Do(m_p(\delta_2), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$$

Then by the IH, we have that

$$M_h \models Do(\delta_1, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \\ Do(\delta_2, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\beta}, S_0))$$

By properties of **setp** and by Lemma A.16, it follows that

$$M_h \models Do(\delta_h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{b}, S_0))$$

□

Lemma A.19 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds. Then, for any high-level program δ_h such that $M_h \models SituationDetermined(\delta_h, do(\vec{\alpha}, S_0))$, we have that*

$$\mathcal{CR}_{M_l}(m_p(\delta_h), do(\vec{a}, S_0)) = \\ \cup_{\vec{\beta} \in \mathcal{CR}_{M_h}(\delta_h, do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0)).$$

Proof (For Golog $^+$ programs)
(⊆) By Lemma A.18.
(⊇) By Lemma A.17.

□

Lemma A.20 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumption 5.1 holds. Then, for any high-level programs δ_h and δ'_h, such that $SituationDetermined(\delta_h, do(\vec{\alpha}, S_0))$ and $SituationDetermined(\delta'_h, do(\vec{\alpha}, S_0))$*
if $\mathcal{CR}_{M_h}(\delta_h, do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\delta'_h, do(\vec{\alpha}, S_0))$, then $\mathcal{CR}_{M_l}(m_p(\delta_h), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(m_p(\delta'_h), do(\vec{a}, S_0))$.

Proof Follows from Lemma A.19. □

Note also that since $m^{-1}_{M_l}$ is a function, the following result follows trivially:

Lemma A.21 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent and that Assumption 5.1 holds. Then for any low-level action sequence $\vec{a}$ such that $M_l \models Do(\text{MONIT}, S_0, do(\vec{a}, S_0))$, it is the case that for any sets of low-level action sequences E_l and E'_l,*
if $E_l \subseteq E'_l$, then $m^{-1}_{M_l}(E_l, s) \subseteq m^{-1}_{M_l}(E'_l, s)$.

Theorem 6.4 If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold, then for any supervision specification represented by a high-level situation-determined program δ^h_{Spec},

$$m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0)), do(\vec{a}, S_0)) = \\ \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)).$$

provided that the sets of action sequences on both sides of this equation have bounded length.

Proof
($\subseteq$) Assume the antecedent. By Theorem 3.1 we have that

$$M_l \models Controllable(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), \text{MONIT}, do(\vec{a}, S_0)).$$

By the definition of mps, we have that

$$\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0))), do(\vec{a}, S_0))) \subseteq \mathcal{CR}_{M_l}(\text{MONIT}, do(\vec{a}, S_0)).$$

Let $E^{mps}_h =$
$$m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0)), do(\vec{a}, S_0)).$$

By Theorem 6.3, we have that

$$M_h \models Controllable(\mathbf{set}(E^{mps}_h), \text{ANY}, do(\vec{a}, S_0)).$$

Also by the definition of mps,

$$\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), do(\vec{a}, S_0)) \\ \subseteq \mathcal{CR}_{M_l}(\text{MONIT} \;\&\; m_p(\delta^h_{Spec}), do(\vec{a}, S_0)).$$

By Assumption 5.1, it follows that $\mathcal{CR}_{M_h}(E^{mps}_h, do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\text{ANY} \;\&\; \delta^h_{Spec}, do(\vec{\alpha}, S_0))$ by Lemma A.21. Then by Theorem 3.1, it follows that $\mathcal{CR}_{M_h}(E^{mps}_h, do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(mps_{\mathit{offl}}((\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$. Thus

$$m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\text{MONIT}, m_p(\delta^h_{Spec}), do(\vec{a}, S_0)), \\ do(\vec{a}, S_0)), do(\vec{a}, S_0)) \subseteq \\ \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)).$$

($\supseteq$) By Theorem 3.1 we have that

$$M_h \models Controllable(mps_{\mathit{offl}}(\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), \text{ANY}, do(\vec{\alpha}, S_0)).$$

Thus by Theorem 6.2, there exists a set of ground action sequences E^{mps}_l such that

- $M_l \models \mathit{Controllable}(\mathbf{set}(E_l^{mps}), \textsc{monit}, do(\vec{a}, S_0))$,
- $m_{M_l}^{-1}(E_l^{mps}, do(\vec{a}, S_0)) =$
  $\mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$, and
- $E_l^{mps} \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$.

By the definition of mps,

$$\mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)) \\ \subseteq \mathcal{CR}_{M_h}(\textsc{any} \ \& \ \delta_{Spec}^h, do(\vec{\alpha}, S_0)).$$

Thus

$$\mathcal{CR}_{M_l}(\mathbf{set}(E_l^{mps}), do(\vec{a}, S_0)) \\ \subseteq \mathcal{CR}_{M_l}(\textsc{monit} \ \& \ m_p(\delta_{Spec}^h), do(\vec{a}, S_0)).$$

by Lemma A.22. By Theorem 3.1 part 3 it follows that $\mathcal{CR}_{M_l}(\mathbf{set}(E_l^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta_{Spec}^h), do(\vec{a}, S_0)), do(\vec{a}, S_0)))$. Thus by Assumption 5.1, it follows that

$$\mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)) \subseteq \\ m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\textsc{monit}, m_p(\delta_{Spec}^h), do(\vec{a}, S_0)), do(\vec{a}, S_0)), do(\vec{a}, S_0)).$$

□

Lemma A.22 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold. Suppose further that for any high-level specification δ_{spec}^h such that $M_h \models \mathit{SituationDetermined}(\delta_h, do(\vec{\alpha}, S_0))$, we have that $E_l^{mps} \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ and that $m_{M_l}^{-1}(E_l^{mps}, do(\vec{a}, S_0)) = \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any},\ \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$. Then, if $\mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\textsc{any} \ \& \ \delta_{Spec}^h, do(\vec{\alpha}, S_0))$, then $\mathcal{CR}_{M_l}(\mathbf{set}(E_l^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit} \ \& \ m_p(\delta_{Spec}^h), do(\vec{a}, S_0))$.*

Proof Take an arbitrary $\vec{c} \in E_l^{mps}$. Since $E_l^{mps} \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$, there exists $\vec{\gamma}$ such that $M_l \models Do(m(\vec{\gamma}), do(\vec{a}, S_0), do(\vec{a}\vec{c}, S_0))$. From this, and since $m_{M_l}^{-1}(E_l^{mps}, do(\vec{a}, S_0)) = \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$, and that $\mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\textsc{any} \ \& \ \delta_{Spec}^h, do(\vec{\alpha}, S_0))$, we have that $M_h \models Do(\delta_{Spec}^h, do(\vec{\alpha}, S_0), do(\vec{\alpha}\vec{\gamma}, S_0))$.
By Lemma A.19 $\mathcal{CR}_{M_l}(m_p(\delta_{Spec}^h), do(\vec{a}, S_0)) = \cup_{\vec{\beta} \in \mathcal{CR}_{M_h}(\delta_{Spec}^h), do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0))$. Thus $\vec{c} \in \mathcal{CR}_{M_l}(m_p(\delta_{Spec}^h), do(\vec{a}, S_0))$.

□

A.3.2 Exploiting Abstraction in Obtaining the MPS

Lemma A.23 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent and that Assumption 5.1 holds. Then for any sequence of ground high-level actions $\vec{\alpha}$ and any sequence of ground low-level actions $\vec{a}$, such that $M_h \models \mathit{Executable}(do(\vec{\alpha}, S_0))$ and $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$, it is the case that $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h)), do(\vec{a}, S_0)) \subseteq E_h$, provided $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h)), do(\vec{a}, S_0)) \subseteq E_h$ is of bounded length.*

Proof By induction on the length of the longest action sequence in $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h, do(\vec{a}, S_0))))$.
Base cases: If $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h)), do(\vec{a}, S_0)) = \emptyset$, the result trivially holds. If $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h)), do(\vec{a}, S_0)) = \{\epsilon\}$, then it must be the case $\epsilon \in E_h$, as the second alternative of $mps_i(E_h)$ starts with

$mps_{\mathit{offl}}(\textsc{oneMonit}, m_p(firsts(E_h)), now)$, whose complete runs must also be compete runs of $\textsc{oneMonit}$ by the definition of mps, and thus none of them can be mapped to ϵ by $m_{M_l}^{-1}$ in $do(\vec{a}, S_0)$.
Inductive Step: Assume that the thesis holds whenever $max_{\vec{\gamma} \in m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h), do(\vec{a}, S_0)))} |\vec{\gamma}| \leq N$. We must show that it must also hold if $max_{\vec{\gamma} \in m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h), do(\vec{a}, S_0)))} |\vec{\gamma}| = N+1$ Assume the antecedent. Take an arbitrary $\vec{\gamma} \in m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h), do(\vec{a}, S_0)))$. If $\vec{\gamma} = \epsilon$, then $\vec{\gamma} \in E_h$ by the same argument as in the analogous base case. Otherwise, it must be the case that there exist $\vec{b}$ and $\vec{c}$ such that $m_{M_l}^{-1}(\vec{b}\vec{c}, do(\vec{a}, S_0)) = \vec{\gamma}$, $\vec{b} \in \mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\textsc{oneMonit}, m_p(firsts(E_h)), now), do(\vec{a}, S_0))$ and $\vec{c} \in \mathcal{CR}_{M_l}(mps_i(rests(E_h, last(m_{M_l}^{-1}(now))))$, $do(\vec{a}\vec{b}, S_0))$. By the definition of MPS, it follows that $Do(\textsc{oneMonit}, do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$. Thus there exist β and $\vec{\gamma}'$ such that $m_{M_l}^{-1}(\vec{b}, do(\vec{a}, S_0)) = \beta$ and $\vec{\gamma} = \beta\vec{\gamma}'$. By Assumption 5.1 we know that $\vec{b}$ maps to a unique high-level action (β). Thus $M_l \models Do(m(\vec{\alpha}\beta), S_0, do(\vec{a}\vec{b}, S_0))$, as well as $M_h \models Executable(do(\vec{\alpha}\beta, S_0))$ by Theorem 5.2. Therefore we can apply the induction hypothesis to obtain that $m_{M_l}^{-1}(\mathcal{CR}_{M_l}(mps_i(rests(E_h, \beta))$, $do(\vec{a}\vec{b}, S_0)), do(\vec{a}\vec{b}, S_0)) \subseteq rests(E_h, \beta)$. Thus $\vec{\gamma}' \in rests(E_h, \beta)$. It follows that $\vec{\gamma} \in E_h$. $\square$

We can show that $mps_i(E_h, do(\vec{a}, S_0))$ is controllable:

Lemma A.24 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $M_h \models \mathcal{C}$ and that Assumptions 5.1 and 6.1 hold. Then for any sequence of ground high-level actions $\vec{\alpha}$ and any sequence of ground low-level actions $\vec{a}$, such that $M_h \models Executable(do(\vec{\alpha}, S_0))$ and $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$, it is the case that $Controllable(mps_i(E_h^{mps}), \textsc{monit}, do(\vec{a}, S_0))$, where $E_h^{mps} = \mathcal{CR}_{M_h}(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$, provided that $m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)))$ has bounded length.*

Proof By induction on the length of longest action sequence in $m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)))$.

Base Cases: If $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) = \emptyset$ the result trivially holds. If $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) = \{\epsilon\}$, then it must be the case $\epsilon \in E_h^{mps}$, as the second alternative of $mps_i(E_h^{mps})$ starts with $mps_{\mathit{offl}}(\textsc{oneMonit}, m_p(firsts(E_h^{mps})), now)$, whose complete runs must also be compete runs of $\textsc{oneMonit}$ by the definition of mps, and thus none of them can be mapped to ϵ by $m_{M_l}^{-1}$ in $do(\vec{a}, S_0)$. Since $\epsilon \in E_h^{mps}$, and by Theorem 3.1 part 2, we have that $Controllable(mps_{\mathit{offl}}(\textsc{any}, \delta_{Spec}^h, do(\vec{\alpha}, S_0)), \textsc{any}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$, this implies that $Controllable(\{\epsilon\}, \textsc{any}, do(\vec{\alpha}, S_0)))$. Thus, by Assumption 6.1, we know that $M_l \models Controllable(\mathbf{set}(\{\epsilon\}), \textsc{monit}, do(\vec{a}, S_0))$, and as a result $M_l \models Controllable(mps_i(E_h^{mps}), \textsc{monit}, do(\vec{a}, S_0))$.

Inductive Step: Assume that the thesis holds for all runs such that $max_{\vec{\gamma} \in m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)))} |\vec{\gamma}| \leq N$. We must show that this holds for N+1 all runs such that $max_{\vec{\gamma} \in m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)))} |\vec{\gamma}| \leq N+1$.

Assume the antecedent. Let $E_l^1 = \{\vec{b} \mid \vec{b} \in \mathcal{CR}_{M_l}(mps_{\mathit{offl}}(\textsc{oneMonit}, m_p(firsts(E_h)), do(\vec{a}, S_0)), do(\vec{a}, S_0))\}$, and for each $\vec{b} \in E_l^1$ let $E_l^{\vec{b}} = \{\vec{c} \mid \vec{c} \in \mathcal{CR}_{M_l}(mps_i(rests(E_h^{mps}, last(m_{M_l}^{-1}(do(\vec{a}\vec{b}, S_0))))), do(\vec{a}\vec{b}, S_0))\}$.

By Assumption 5.1 we know that each $\vec{b}$ and $\vec{c}$ maps to a unique (sequence of) high-level action(s). By induction hypothesis, we have that $M_l \models Controllable(mps_i(rests(E_h^{mps}, last(m_{M_l}^{-1}(do(\vec{a}\vec{b}, S_0))))), do(\vec{a}\vec{b}, S_0))$, thus, $M_l \models Controllable(\mathbf{set}(E_l^{\vec{b}}), \textsc{monit}, do(\vec{a}\vec{b}, S_0))$. By Theorem 3.1, we have that $M_l \models Controllable(\mathbf{set}(E_l^1), \textsc{oneMonit}, do(\vec{a}, S_0))$. Let $E_l = \{\vec{b}\vec{c} \mid \vec{b} \in E_l^1$ and $\vec{c} \in E_l^{\vec{b}}\} \cup Q$ where $Q = \{\epsilon\}$ if $\epsilon \in E_h^{mps}$ and $Q = \emptyset$ otherwise. It follows by Lemma A.9 that $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$. Thus, $M_l \models Controllable(mps_i(E_h^{mps}), \textsc{monit}, do(\vec{a}, S_0))$. $\square$

We can show that the hierarchically synthesized MPS $mps_i(E_h^{mps})$ includes any controllable set of low-level action sequences that satisfies high-level specification:

Lemma A.25 *Suppose that we have bisimilar models $M_h \sim_m M_l$ for an agent, $M_h \models \mathcal{C}$ and that Assumptions 5.1 and 6.1 hold. Then, for any sequence of ground high-level actions $\vec{\alpha}$ and any sequence of ground* *low-level actions $\vec{a}$ such that $M_h \models Executable(do(\vec{\alpha}, S_0))$ and $M_l \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$, it is the case that*

for any supervision specification represented by a high-level situation-determined program δ^h_{Spec} and for any set of ground low-level action sequences E_l such that $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\text{MONIT}, do(\vec{a}, S_0))$,

if $M_l \models Controllable(\mathbf{set}(E_l), \text{MONIT}, do(\vec{a}, S_0))$ and $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))$ has bounded length, then $E_l \subseteq \mathcal{CR}_{M_l}(mps_i(E^{mps}_h), do(\vec{a}, S_0)))$, where $E^{mps}_h = \mathcal{CR}_{M_h}(mps_{offl}(\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$.

Proof By induction on the length of the longest action sequence in $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))$.
Base Cases: In the case where $E_l = \emptyset$, the result trivially holds. In the case where $E_l = \{\epsilon\}$, then $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = \{\epsilon\}$. Since $M_l \models Controllable(\mathbf{set}(E_l), \text{MONIT}, do(\vec{a}, S_0))$, we also have that $M_h \models Controllable(\mathbf{set}(\epsilon), \text{ANY}, do(\vec{\alpha}, S_0)$ by Theorem 6.3. Since $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$, we have that $M_l \models Final(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$, and it follows that $M_h \models Final(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$ by Corollary A.15. Therefore we have that $\epsilon \in \mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$, as well as $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$. Clearly, it is also the case that $\{\epsilon\} \subseteq \mathcal{CR}_{M_h}(\text{ANY}, do(\vec{\alpha}, S_0))$. It then follows by Theorem 3.1 part 3 that $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) \subseteq E^{mps}_h$.

Induction Step: Assume that the result holds for any E_l such that $max_{\vec{b} \in m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))}|\vec{b}| = K$ (IH). Let's show that it must hold for any E_l such that $max_{\vec{b} \in m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))}|\vec{b}| = K + 1$. Assume the antecedent. Let $E_h = m^{-1}_{M_l}(E_l, do(\vec{a}, S_0))$. By Theorem 6.3, we have that $M_h \models Controllable(\mathbf{set}(E_h), \text{ANY}, do(\vec{\alpha}, S_0))$.

We have that $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$. By Lemma A.19, we have that $\mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) = \cup_{\vec{\beta} \in \mathcal{CR}_{M_h}(\delta^h_{Spec}), do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0))$. Thus, by Assumption 5.1, we have that every $\vec{b} \in \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$ maps to a unique $\vec{\beta} \in \mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$. By Assumption 5.1, and the fact that $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$, it follows that $E_h \subseteq \mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$. Thus by Theorem 3.1 part 3, we have that $E_h \subseteq E^{mps}_h$. It follows trivially that $firsts(E_h) \subseteq firsts(E^{mps}_h)$. Let $E^1_l = \{\vec{b} \mid \vec{b}\vec{c} \in E_l$ for some $\vec{c}$ and $M_l \models Do(\text{ONEMONIT}, do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))\}$.

Since $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = E_h$ we have that $m^{-1}_{M_l}(E^1_l, do(\vec{a}, S_0)) = firsts(E_h)$. Thus we have $m^{-1}_{M_l}(E^1_l, do(\vec{a}, S_0)) \subseteq firsts(E^{mps}_h)$. By Lemma A.19, we have that $\mathcal{CR}_{M_l}(m_p(firsts(E_{mps_h}), do(\vec{a}, S_0)) = \cup_{\beta \in \mathcal{CR}_{M_h}(firsts(E_{mps_h}, do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\beta), do(\vec{a}, S_0))$. By Assumption 5.1 we know that every $\vec{b} \in E^1_l$, there is a unique β such that $m^{-1}_{M_l}(\vec{b}) = \beta$. Thus $\vec{b} \in \cup_{\beta \in \mathcal{CR}_{M_h}(firsts(E_{mps_h}, do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\beta), do(\vec{a}, S_0))$. Thus it follows that $E^1_l \subseteq \mathcal{CR}_{M_l}(m_p(firsts(E_{mps_h})), do(\vec{a}, S_0))$.

Since $M_l \models Controllable(\mathbf{set}(E_l), \text{MONIT}, do(\vec{a}, S_0))$, we have that $M_l \models Controllable(\mathbf{set}(E^1_l), \text{ONEMONIT}, do(\vec{a}, S_0))$ by Lemma A.10. By the definition of E^1_l, we have that $E^1_l \subseteq \mathcal{CR}_{M_l}(\text{ONEMONIT}, do(\vec{a}, S_0))$. Thus by Theorem 3.1 part 3, we have that $E^1_l \subseteq \mathcal{CR}_{M_l}(mps(\text{ONEMONIT}, m_p(firsts(E_{mps_h})), do(\vec{a}, S_0)), do(\vec{a}, S_0))$.

We also have that for every $\vec{b} \in E^1_l$, $\beta_{\vec{b}}$ and $\delta_{\vec{b}}$ such that $M_l \models Do(m(\beta_{\vec{b}}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$ ($\vec{b}$ maps to a unique $\beta_{\vec{b}}$ by Assumption 5.1), $M_h \models Trans(\delta^h_{Spec}, do(\vec{\alpha}, S_0), \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0))$, and $E_{\vec{b}} = \{\vec{c} \mid \vec{b}\vec{c} \in E_l\}$:

1. $M_l \models Controllable(\mathbf{set}(E_{\vec{b}}), \text{MONIT}, do(\vec{a}\vec{b}, S_0))$, since $M_l \models Controllable(\mathbf{set}(E_l), \text{MONIT}, do(\vec{a}, S_0))$.

2. $E_{\vec{b}} \subseteq \mathcal{CR}_{M_l}(m_p(\delta_{\vec{b}}), do(\vec{a}\vec{b}, S_0) \subseteq \mathcal{CR}_{M_l}(\text{MONIT}, do(\vec{a}\vec{b}, S_0))$,
since $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\text{MONIT}, do(\vec{a}, S_0))$.

3. $rests(E^{mps}_h, \beta_{\vec{b}}) =$
$\mathcal{CR}_{M_h}(mps_{offl}(\text{ANY}, \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)), do(\vec{\alpha}\beta_{\vec{b}}, S_0))$, since $E^{mps}_h = \mathcal{CR}_{M_h}(mps_{offl}(\text{ANY}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$ by Lemma A.26.

Thus we can apply the induction hypothesis and get that $E_{\vec{b}} \subseteq \mathcal{CR}_{M_l}(mps_i(rests(E_{mps_h}, \beta_{\vec{b}})), do(\vec{a}\vec{b}, S_0))$.

It follows that $E_l \subseteq \mathcal{CR}_{M_l}(mps_i(E_{mps_h})), do(\vec{a}, S_0))$.

If $\epsilon \in E_l$, then $m^{-1}_{M_l}(E_l, do(\vec{a}, S_0)) = \{\epsilon\}$. Since $M_l \models Controllable(\mathbf{set}(E_l), \textsc{monit}, do(\vec{a}, S_0))$, we also have that $M_h \models Controllable(\mathbf{set}(\epsilon), \textsc{any}, do(\vec{\alpha}, S_0)$ by Theorem 6.3, and $\epsilon \in E_h$. Thus $\epsilon \in \mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0))$ by a similar argument as the base case. □

Lemma A.26 *Suppose $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold. Suppose further that $E_h^{mps} = \mathcal{CR}_{M_h}(mps_{offl}(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0)), do(\vec{\alpha}, S_0))$, $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$, $E_l^1 = \{\vec{b} \mid \vec{b}\vec{c} \in E_l$ for some $\vec{c}$ and $M_l \models Do(\textsc{oneMonit}, do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))\}$. Then for every $\vec{b} \in E_l^1$, $\beta_{\vec{b}}$ and $\delta_{\vec{b}}$ such that $M_l \models Do(m(\beta_{\vec{b}}), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$, $M_h \models Trans(\delta^h_{Spec}, do(\vec{\alpha}, S_0), \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0))$, and $E_{\vec{b}} = \{\vec{c} \mid \vec{b}\vec{c} \in E_l\}$, we have that $rests(E_h^{mps}, \beta_{\vec{b}}) = \mathcal{CR}_{M_h}(mps_{offl}(\textsc{any}, \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)), do(\vec{\alpha}\beta_{\vec{b}}, S_0))$.*

Proof

($\subseteq$) Suppose $E_h^{\beta_{\vec{b}}} = \{\vec{\gamma} \mid \beta_{\vec{b}}\vec{\gamma} \in E_h^{mps}\}$. We have that $E_h^{\beta_{\vec{b}}} \subseteq \mathcal{CR}_{M_h}(\delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)) \subseteq \mathcal{CR}_{M_h}(\textsc{any}, do(\vec{\alpha}\beta_{\vec{b}}, S_0))$. Since E_h^{mps} is controllable wrt $\textsc{any}$, by Theorem 3.1, part 3, we have that $M_h \models Controllable(E_h^{\beta_{\vec{b}}}, \textsc{any}, do(\vec{\alpha}\beta_{\vec{b}}, S_0))$. Thus $E_h^{\beta_{\vec{b}}} \subseteq \mathcal{CR}_{M_h}(mps_{offl}(\textsc{any}, \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)), do(\vec{\alpha}\beta_{\vec{b}}, S_0))$.

($\supseteq$) By contradiction. Assume the antecedent. Let $E'_h = \mathcal{CR}(mps_{offl}(\textsc{any}, \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)), do(\vec{\alpha}\beta_{\vec{b}}, S_0))$. Assume that $rests(E_h^{mps}, \beta_{\vec{b}}) \not\supseteq E'_h$. Let

$$E_h = \{\vec{\gamma} \mid \vec{\gamma} \in E_h^{mps} \text{ and } \vec{\gamma} \neq \beta_{\vec{b}}\vec{\gamma}' \text{ for some } \vec{\gamma}'\} \cup \{\beta_{\vec{b}}\vec{\gamma}'' \mid \vec{\gamma}'' \in E'_h\}$$

Since $M_h \models Controllable(set(E_h^{mps}), \textsc{any}, do(\vec{\alpha}, S_0))$ and $M_h \models Controllable(mps_{offl}(\textsc{any}, \delta_{\vec{b}}, do(\vec{\alpha}\beta_{\vec{b}}, S_0)), \textsc{any}, do(\vec{\alpha}\beta_{\vec{b}}, S_0))$, it is easy to show that $M_h \models Controllable(set(E_h), \textsc{any}, do(\vec{\alpha}, S_0))$ (by an argument similar to that used in Lemma A.9). It is also easy to show that $E_h \subseteq \mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\textsc{any}, do(\vec{\alpha}, S_0))$. Then by Theorem 3.1 part 3, it follows that $E_h \subseteq E_h^{mps}$. Thus $rests(E_{mps_h}, \beta_{\vec{b}}) \supseteq E'_h$, contradiction.

□

Theorem 6.5 If $M_h \sim_m M_l$ and $\vec{a}$ is an m-refinement of an executable $\vec{\alpha}$, $M_h \models \mathcal{C}$, and Assumptions 5.1 and 6.1 hold, then for any supervision specification represented by a high-level situation-determined program δ^h_{Spec} where $\mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0))$ and $m^{-1}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0))$ have bounded length,

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

Proof

($\subseteq$) By Lemma A.24, we have that $Controllable(mps_i(E_h^{mps}), \textsc{monit}, do(\vec{a}, S_0))$. We then need to show that $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(m_p(\delta_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$. We have that $E_h^{mps} \subseteq \mathcal{CR}_{M_h}(\delta^h_{Spec}\ \&\ \textsc{any}, do(\vec{\alpha}, S_0))$, and $\mathcal{CR}_{M_h}(\delta^h_{Spec}, do(\vec{\alpha}, S_0)) \subseteq \mathcal{CR}_{M_h}(\textsc{any}, do(\vec{\alpha}, S_0))$.

By Assumption 5.1 we have that for each $\vec{b} \in \mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0))$, there is a unique $\vec{\beta} \in m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps})), do(\vec{a}, S_0))$. By Lemma A.23, we know that $m^{-1}_{M_l}(\mathcal{CR}_{M_l}(mps_i(E_h^{mps})), do(\vec{a}, S_0)) \subseteq E_h^{mps}$. By Lemma A.19, $\cup_{\vec{\beta} \in E_h^{mps}} \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0)) = \mathcal{CR}_{M_l}(m_p(mps_{offl}(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0))), do(\vec{a}, S_0))$. Thus for every $\vec{b} \in \mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0))$, we have $\vec{b} \in \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0))$ where $\vec{\beta} \in E_h^{mps}$. Therefore, $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(m_p(mps_{offl}(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0))), do(\vec{a}, S_0))$

By Lemma A.20 we have that $\mathcal{CR}_{M_l}(m_p(mps_{offl}(\textsc{any}, \delta^h_{Spec}, do(\vec{\alpha}, S_0))), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$. Thus $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0))$. By definition of $\textsc{monit}$, Theorem 5.2, and Lemmas A.20 and A.19 we have that $\mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$. Therefore, we have that $\mathcal{CR}_{M_l}(mps_i(E_h^{mps}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$ The result follows by Theorem 3.1 part 3.

($\supseteq$) By Lemma A.25, we have that $mps_i(E_h^{mps})$ includes any controllable (wrt MONIT) set E_l such that $E_l \subseteq \mathcal{CR}_{M_l}(m_p(\delta^h_{Spec}), do(\vec{a}, S_0)) \subseteq \mathcal{CR}_{M_l}(\textsc{monit}, do(\vec{a}, S_0))$. The result follows by Theorem 3.1 part 3.

A.4 Abstracting Online Agent Behavior

A.4.1 Sound Abstraction in Online Executions

Theorem 7.1 If $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to refinement mapping m and $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))$ holds, then $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to m.

Proof Take an arbitrary model M_l of $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\}$. We need to show that there is a model of $D_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ which is bisimilar to M_l. Since D_h is a sound abstraction of D_l relative to mapping m, we know that there exists a model M_h of D_h such that $M_h \sim_m M_l$. By the condition in the antecedent that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$, we know that $M_l \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$. By Theorem 5.2 we have that $M_h \models \{Executable(do(\vec{\alpha}, S_0))\}$, and thus $M_h \models D_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$. □

Theorem 7.2 Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m, $\mathcal{D}_l \cup C \cup \{Do(m(\vec{\alpha}), S_0, do(\vec{a}, S_0))\}$ is satisfiable, and $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$ for some ground high-level action sequence $\vec{\alpha}$ and ground low-level action sequence $\vec{a}$. Then we have that for any ground high-level action sequence $\vec{\beta}$ and high-level situation-suppressed formula ϕ, if

$$\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \phi[do(\vec{\alpha}\vec{\beta}, S_0)],$$

then

$$\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \exists s. Do(m(\vec{\beta}), do(\vec{a}, S_0), s) \wedge m(\phi)[s].$$

Proof By Contradiction. Assume the antecedent holds and the consequent does not. Thus there is a model M_l of $\mathcal{D}_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\}$, where $\neg\exists s. Do(m(\vec{\beta}), do(\vec{a}, S_0), s)) \wedge \neg m(\phi)[s]$ is satisfiable. Since $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ wrt m, and $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models Do(m(\vec{\alpha}), S_0, do(S_0, \vec{a}))$, then by Theorem 7.1, we have that $D_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $D_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to m. Thus by Theorem 5.2, there is a model M_h of $D_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ that is bisimilar to M_l and satisfies $\{\neg Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \neg\phi[do(\vec{\alpha}\vec{\beta}, S_0)]$. That means $D_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \cup \{\neg Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \neg\phi[do(\vec{\alpha}\vec{\beta}, S_0)]\}$ is satisfiable. However, this contradicts the assumption that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Executable(do(\vec{\alpha}\vec{\beta}, S_0)) \wedge \phi[do(\vec{\alpha}\vec{\beta}, S_0)]$. □

A.4.2 Online Executions

Lemma A.27 *If $\mathcal{D}$ is satisfiable, for all δ^i,δ, ground action sequence $\vec{a}$, if $\langle\delta^i, \epsilon\rangle \rightarrow^*_{\vec{a}} \langle\delta, \vec{a}\rangle$ and $\langle\delta, \vec{a}\rangle^{\checkmark}$ then $\mathcal{D} \cup \mathcal{C} \cup \{Trans^*(\delta^i, S_0, \delta, do(\vec{a}, S_0)) \cup Final(\delta, do(\vec{a}, S_0))\}$ is satisfiable.*

Proof By induction on the length of $\vec{a}$.

Base Case: $\vec{a} = \epsilon$. By definition, $\langle\delta^i, \epsilon\rangle^{\checkmark}$ if and only if $\mathcal{D} \cup \mathcal{C} \models Final(\delta^i, S_0)$. Thus, in online executions, since a configuration is final if the agent *knows* that is final, and since $\mathcal{D}$ is satisfiable, thus, $Final(\delta^i, S_0)$ is satisfiable.

Induction Step: Assume the claim holds for any $\vec{a}$ where length of $\vec{a} = K$. We need to show that the claim holds for any $\vec{a}$ where length of $\vec{a} = K + 1$. Assume that the antecedent holds. Assume $\vec{a} = \vec{a'}b$. There are two cases:

[Case 1] When $b \in A^o$. By definition of online transition, the agent can only execute b if the theory entails that an online transition involving action b exists. Since $\mathcal{D}$ is satisfiable, then there is at least one model $\mathcal{D}$ where $\mathcal{D} \cup \mathcal{C} \cup \{Trans^*(\delta^i, S_0, \delta, do(\vec{a}b, S_0))$ is satisfiable.

[Case 2] When $b \in A^e$. By definition of online transition, the agent can only execute b if it is satisfiable that an online transition involving action b exists. Thus, there is at least one model $\mathcal{D}$ where $\mathcal{D} \cup \mathcal{C} \cup \{Trans^*(\delta^i, S_0, \delta, do(\vec{a}b, S_0))$ is satisfiable.

In both cases 1 and 2, we have that by definition, $\langle \delta, \vec{a} \rangle^{\checkmark}$ if and only if $\mathcal{D} \cup \mathcal{C} \models Final(\delta, do(\vec{a}, S_0))$. Thus, in online executions, since a configuration is final if the agent *knows* that is final, and since $\mathcal{D}$ is satisfiable, thus, $Final(\delta, do(\vec{a}, S_0))$ is satisfiable. □

Lemma A.28 *If m is a refinement mapping from $\mathcal{D}_h$ to $\mathcal{D}_l$ and for all ground high-level action sequence $\vec{\alpha}$, ground low-level action sequence $\vec{a_l}$, there exists δ_m such that $\langle m(\vec{\alpha}), \epsilon \rangle \rightarrow^*_{\vec{a_l}} \langle \delta_m, \vec{a_l} \rangle$ and $\langle \delta_m, \vec{a_l} \rangle^{\checkmark}$ then $\mathcal{D}_l \cup \mathcal{C} \cup \{Do(m(\vec{\alpha}), S_0, do(\vec{a_l}, S_0))\}$ is satisfiable.*

Proof Follows from Lemma A.27. □

A.4.3 Hierarchical Contingent Planning

We now define the *length of a strategy*, and the abbreviation **seq**, which allows us to sequentially compose strategies. These are used in the following results.

Length of a Strategy. Let us define the length of a strategy γ, $length(\gamma)$, as follows:

(a) when $\gamma = nil$, $length(\gamma) = 0$.

(b) when $\gamma = \alpha; \gamma'$, $length(\alpha; \gamma') = 1 + length(\gamma')$.

(c) when $\gamma = \mathbf{set}(H)$, where $H = \{\alpha_1; \gamma'_1, \ldots; \alpha_n; \gamma'_n\}$, $length(\mathbf{set}(H)) = max(length(\gamma'))$, where $\gamma' \in H$

The seq Abbreviation. We also define an abbreviation that allows us to sequentially compose strategies. $\mathbf{seq}(\gamma, G)$, where γ is a strategy and G is a set of pairs $\langle \vec{a}, \gamma_{\vec{a}} \rangle$, stands for the strategy that extends any branch of γ that does $\vec{a}$ by $\gamma_{\vec{a}}$ for all $\langle \vec{a}, \gamma_{\vec{a}} \rangle \in G$:

$$\mathbf{seq}(\gamma, G) = \begin{cases} nil \text{ if } \gamma = nil \text{ and } \langle \epsilon, \gamma_\epsilon \rangle \notin G \\ \gamma_\epsilon \text{ if } \gamma = nil \text{ and } \langle \epsilon, \gamma_\epsilon \rangle \in G \\ \\ a; \mathbf{seq}(\gamma', G') \text{ if } \gamma = a; \gamma' \\ \quad \text{where } G' = \{\langle \vec{b}, \gamma_{\vec{b}} \rangle \mid \langle \vec{a}, \gamma_{\vec{a}} \rangle \in G \text{ and } \vec{a} = a; \vec{b} \text{ and } \gamma_{\vec{b}} = \gamma_{\vec{a}}\} \\ \\ \mathbf{set}(H') \text{ if } \gamma = \mathbf{set}(H) \\ \quad \text{where } H' = \{b; \mathbf{seq}(\gamma_b, G_b) \mid b; \gamma_b \in H \text{ and} \\ \qquad\qquad G_b = \{\langle \vec{c}, \gamma_{\vec{c}} \rangle \mid \langle \vec{a}, \gamma_{\vec{a}} \rangle \in G \text{ and } \vec{a} = b; \vec{c} \text{ and } \gamma_{\vec{c}} = \gamma_{\vec{a}}\}\} \end{cases}$$

Lemma A.29
Suppose that Assumption 7.1 holds. Then for any online situation-determined programs δ_1 and δ_2, ground action sequence $\vec{a}$, and strategy γ_1, if $AbleBy(\delta_1, \vec{a}, \gamma_1)$ and for all $\vec{b_i}$ such that $\langle \gamma_1, \vec{a} \rangle \rightarrow^*_{\vec{b_i}} \langle \gamma_{1,i}, \vec{a}\vec{b_i} \rangle$ and $\langle \gamma_{1,i}, \vec{a}\vec{b_i} \rangle^{\checkmark}$ for some $\gamma_{1,i}$, there exists $\gamma_{\vec{b_i}}$ such that $AbleBy(\delta_2, \vec{a}\vec{b_i}, \gamma_{\vec{b_i}})$, then $AbleBy(\delta_1; \delta_2, \vec{a}, \mathbf{seq}(\gamma_1, E))$, where $E = \{\langle \vec{b_i}, \gamma_{\vec{b_i}} \rangle \mid \langle \gamma_1, \vec{a} \rangle \rightarrow^*_{\vec{b_i}} \langle \gamma_{1,i}, \vec{a}\vec{b_i} \rangle$ and $\langle \gamma_{1,i}, \vec{a}\vec{b_i} \rangle^{\checkmark}$ for some $\gamma_{1,i}$ and $AbleBy(\delta_2, \vec{a}\vec{b_i}, \gamma_{\vec{b_i}})\}$.

Proof By induction on $length(\gamma_1)$.

Base Case, where $length(\gamma_1) = 0$: Then we have $\gamma_1 = nil$, and the antecendent implies that there exists γ_ϵ such that $AbleBy(\delta_2, \vec{a}, \gamma_\epsilon)$. It follows that $AbleBy(\delta_1; \delta_2, \vec{a}, \mathbf{seq}(\gamma_1, \gamma_\epsilon))$.

Induction Step: Assume the claim holds for any γ_1 where $length(\gamma_1) \leq K$ (IH). We need to show that the claim holds for any γ_1 where $length(\gamma_1) = K + 1$. Assume that the antecedent holds. There are two cases:

[Case 1]: where $\gamma_1 = a; \gamma'_1$ and $a \in \mathcal{A}^o$. Since we have $AbleBy(\delta_1, \vec{a}, \gamma_1)$ it follows that there exists δ'_1 such that $\langle \delta_1, \vec{a} \rangle \rightarrow_a \langle \delta'_1, \vec{a}a \rangle$ and $AbleBy(\delta'_1, \vec{a}a, \gamma'_1)$. From the antecedent, it follows that for all $\vec{b'_i}$ such that $\langle \gamma_1', \vec{a}a\rangle \rightarrow^*_{\vec{b_i'}} \langle \gamma_{1,i}, \vec{a}a\vec{b_i'}\rangle$ and $\langle \gamma_{1,i}, \vec{a}a\vec{b_i'}\rangle^{\checkmark}$ for some $\gamma_{1,i}$, there exists $\gamma_{\vec{b_i'}}$ such that $AbleBy(\delta_2, \vec{a}a\vec{b_i'}, \gamma_{\vec{b_i'}})$. By the induction hypothesis (IH), we then have that $AbleBy(\delta_1'; \delta_2, \vec{a}a, \mathbf{seq}(\gamma_1', E')))$ where $E' = \{\langle \vec{b_i'}, \gamma_{\vec{b_i'}}\rangle \mid \langle \gamma_1, \vec{a}a\rangle \rightarrow^*_{\vec{b_i'}} \langle \gamma_{1,i}, \vec{a}a\vec{b_i'}\rangle$ and $\langle \gamma_{1,i}, \vec{a}\vec{b_i}\rangle^{\checkmark}$ for some $\gamma_{1,i}$ and $AbleBy(\delta_2, \vec{a}a\vec{b_i}, \gamma_{\vec{b_i'}})\}$. Then by part (B) of the definition of *AbleBy*, the result follows.

[Case 2]: where $\gamma_1 = \mathbf{set}(E)$ where E is a non-empty set of programs of the form $a_i; \gamma_{1,i}'$ where $a_i \in \mathcal{A}^e$ and $\gamma_{1,i}'$ is a strategy that follows a_i. Since we have $AbleBy(\delta_1, \vec{a}, \gamma_1)$ it follows that there exists an exogenous action a, a strategy γ_1' and program δ_1' such that $\langle \delta_1, \vec{a}\rangle \rightarrow_a \langle \delta_1', \vec{a}a\rangle$ and $AbleBy(\delta_1', \vec{a}a, \gamma_1')$. Now take an arbitrary a_i and $\delta_{1,i}'$ such that $\langle \delta_1, \vec{a}\rangle \rightarrow_{a_i} \langle \delta_{1,i}', \vec{a}a_i\rangle$. By the definition of *AbleBy*, there exists a $\gamma_{1,i}'$ such that $AbleBy(\delta_{1,i}', \vec{a}a_i, \gamma_{1,i}')$. From the antecedent, it follows that for all $\vec{b_i'}$ such that $\langle \gamma_{1,i}', \vec{a}a_i\rangle \rightarrow^*_{\vec{b_i'}} \langle \gamma_{1,i}'', \vec{a}a_i\vec{b_i'}\rangle$ and $\langle \gamma_{1,i}'', \vec{a}a\vec{b_i'}\rangle^{\checkmark}$ for some $\gamma_{1,i}''$, there exists $\gamma_{\vec{b_i'}}$ such that $AbleBy(\delta_2, \vec{a}a_i\vec{b_i'}, \gamma_{\vec{b_i'}})$. By the IH, we then have that $AbleBy(\delta_{1,i}'; \delta_2, \vec{a}a_i, \mathbf{seq}(\gamma_{1,i}', E'))$ where $E' = \{\langle \vec{b_i'}, \gamma_{\vec{b_i'}}\rangle \mid \langle \gamma_{1,i}', \vec{a}a_i\rangle \rightarrow^*_{\vec{b_i'}} \langle \gamma_{1,i}'', \vec{a}a_i\vec{b_i'}\rangle$ and $\langle \gamma_{1,i}'', \vec{a}a_i\vec{b_i}\rangle^{\checkmark}$ for some $\gamma_{1,i}''$ and $AbleBy(\delta_2, \vec{a}a_i\vec{b_i}, \gamma_{\vec{b_i'}})\}$. Then by part (C) of the definition of *AbleBy*, the result follows.

□

We can show that if $NecTerminates(\delta, \vec{a})$, then the length of any online execution from configuration $\langle \delta, \vec{a}\rangle$ is bounded:

Lemma A.30 *For any program δ and any ground action sequence $\vec{a}$, if $NecTerminates(\delta, \vec{a})$, then there exists $K \in \mathbb{N}$ such that for all $\delta', \vec{b}$ $\langle \delta, \vec{a}\rangle \rightarrow^*_{\vec{b}} \langle \delta', \vec{a}\vec{b}\rangle$, $|\vec{b}| \leq K$.*

Proof Suppose we have an infinite length online execution from $\langle \delta, \vec{a}\rangle$ such that $\langle \delta, \vec{a}\rangle$, $\langle \delta_1, \vec{a}b_1\rangle$,$\langle \delta_2, \vec{a}b_1b_2\rangle$, ... Then none of the configurations in it is such that there is no further online transitions from it, so it is not in $NecTerminates(\delta_i, \vec{a}b_i)$ by part A of definition of *NecTerminates*. Also, no configuration in the execution is in $NecTerminates(\delta_i, \vec{a}b_i)$ by part B of the definition as they all have a successor which is not in *NecTerminates*. Thus $\neg NecTerminates(\delta, \vec{a})$. □

Lemma A.31 *Suppose that Assumption 7.1 holds. Then for any online situation-determined program δ and any ground action sequence $\vec{a}$, if we have $NecTerminates(\delta, \vec{a})$, then there exists a strategy γ such that $AbleBy(\delta, \vec{a}, \gamma)$.*

Proof By Lemma A.30 we know that length of online executions from $\langle \delta, \vec{a}\rangle$ is bounded. Thus, we can prove this Lemma by induction on the maximum length of online executions starting in $\langle \delta, \vec{a}\rangle$.

Base case: When the maximum length of online executions from $\langle \delta, \vec{a}\rangle$ is 0. Then we have $\langle \delta, \vec{a}\rangle^{\checkmark}$. Thus we can choose $\gamma = nil$ such that we have $AbleBy(\delta, \vec{a}, nil)$.

Induction Step: Assume that the claim holds for any $\langle \delta, \vec{a}\rangle$ with a maximum length of online executions less than or equal to K (IH). We need to show that the claim holds for any $\langle \delta, \vec{a}\rangle$ with a maximum length of online executions equal to $K + 1$. Assume that the antecedent holds.

Since the maximum length is ≥ 1, there exists a and δ' such that such that $\langle \delta, \vec{a}\rangle \rightarrow_a \langle \delta', \vec{a}a\rangle$.

Case 1 $a \in A^o$. By the definition of *NecTerminates*, we have that there exists a program δ' such that $\langle \delta, \vec{a}\rangle \rightarrow_a \langle \delta', \vec{a}a\rangle$ and that $NecTerminates(\delta', \vec{a}a)$. By (IH) we have there exists γ' such that $AbleBy(\delta', \vec{a}a; \gamma')$. Then by the definition of *AbleBy* part B, we have that $AbleBy(\delta, \vec{a}, a; \gamma)$.

Case 2 $a \in A^e$. By the definition of *NecTerminates*, we have for all a_i, δ_i' such that $\langle \delta, \vec{a}\rangle \rightarrow_{a_i} \langle \delta_i', \vec{a}a_i\rangle$ we have that $NecTerminates(\delta_i', \vec{a}a_i)$. By (IH) for all such a_i, δ_i' there exists γ_i such that $AbleBy(\delta_i', \vec{a}a_i; \gamma_i')$. By the definition of *AbleBy* part C, we have $AbleBy(\delta, \vec{a}, \gamma)$ where $\gamma = \mathbf{set}(G)$, where $G = \{\langle a_i; \gamma_i\rangle \mid$ there exists δ_i' such that $\langle \delta, \vec{a}\rangle \rightarrow_{a_i} \langle \delta_i', \vec{a}a_i\rangle$ and $AbleBy(\delta_i', \vec{a}a_i, \gamma_i)\}$.

□

Theorem 7.5 Suppose that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0))\}$ is a sound abstraction of $\mathcal{D}_l \cup \{Executable(do(\vec{a}, S_0))\}$ relative to mapping m and Assumptions 7.1, 7.2, 7.3, and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle\vec{\alpha}, \epsilon\rangle \rightarrow^*_{\vec{\alpha}} \langle\delta'_h, \vec{\alpha}\rangle$ for some δ'_h and $\langle\delta'_h, \vec{\alpha}\rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle\delta_l, \vec{a}\rangle$ and $\langle\delta_l, \vec{a}\rangle^{\checkmark}$ for some δ_l if there exists a ground high-level action $\beta \in \mathcal{A}^e$ such that $D_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta, do(\vec{\alpha}, S_0))\}$ is satisfiable, then $NecTerminates$(ANYONEEXOHL, $\vec{a}$).

Proof Assume the antecedent. We have that there exists a ground high-level action $\beta \in \mathcal{A}^e$ such that $\mathcal{D}_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta, do(\vec{\alpha}, S_0))\}$ is satisfiable. From this, it follows by Assumptions 7.1 (Turn-Taking) and 7.3 (Always Known Whose Turn It Is) that there does not exist a high-level ordinary action $\beta' \in \mathcal{A}^o_h$ such that $D_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta', do(\vec{\alpha}, S_0))\}$ is satisfiable. Since we have a sound abstraction, it then follows by Theorem 5.2 that there does not exist a high-level ordinary action $\beta' \in \mathcal{A}^o_h$ such that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta'), do(\vec{a}, S_0), s)\}$ is satisfiable. From this, it follows by Assumption 7.2 (Non-Blocking) that there exists an exogenous high-level action $\beta \in \mathcal{A}^e_h$ such that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta), do(\vec{a}, S_0), s)\}$ is satisfiable. Thus by Assumption 7.6 (Exogenous HL Actions Never Block), we have that $NecTerminates$(ANYONEEXOHL, $\vec{a}$), and furthermore by Lemma A.31, we have that there exists a low-level strategy γ_1 such that $AbleBy$(ANYONEEXOHL, $\vec{a}, \gamma_1$).

□

Theorem 7.6 Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m and Assumptions 5.2, 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle\vec{\alpha}, \epsilon\rangle \rightarrow^*_{\vec{\alpha}} \langle\delta'_h, \vec{\alpha}\rangle$ for some δ'_h and $\langle\delta'_h, \vec{\alpha}\rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle\delta_l, \vec{a}\rangle$ and $\langle\delta_l, \vec{a}\rangle^{\checkmark}$ for some δ_l and for any online situation-determined high-level program δ_h and high-level strategy γ_h, if $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$, then there exists a low-level strategy γ_l such that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$.

Proof By induction on $length(\gamma_h)$.

Base case: When $length(\gamma_h) = 0$, then $\gamma_h = nil$. Thus by part (A) of the definition of $AbleBy$, we have that $\langle\delta_h, \vec{\alpha}\rangle^{\checkmark}$. We have that $m_p(\gamma_h) = m_p(nil) = nil$, which is always online "final". Let $\gamma_l = nil$. Then by part (A) in the definition of $AbleBy$, we have that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$.

Induction Step: Assume that the claim holds for any γ_h where $length(\gamma_h) = K$. We need to show that the claim holds for any γ_h of length $K + 1$. Assume that the antecedent holds. Then we have two cases.

Case 1: $\gamma_h = \beta; \gamma'_h$ where $\beta \in \mathcal{A}^o$.
Since we have $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$, it then follows that there exists δ'_h such that $\langle\delta_h, \vec{\alpha}\rangle \rightarrow_\beta \langle\delta'_h, \vec{\alpha}\beta\rangle$ and $AbleBy(\delta'_h, \vec{\alpha}\beta, \gamma'_h)$. The former implies that $D_h \cup \{Executable(do(\vec{\alpha}, S_0))\} \models Poss(\beta, do(\vec{\alpha}, S_0))$. It follows by Assumption 7.4 and Theorem 7.2 that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0))\} \models \exists s.Do(m(\beta), do(\vec{a}, S_0), s)$. By Assumption 7.5, it then follows that there exists a strategy γ_β such that $AbleBy(m(\beta), \vec{a}, \gamma_\beta)$. Take an arbitrary $\vec{b}$ such that $\langle\gamma_\beta, \vec{a}\rangle \rightarrow^*_{\vec{b}} \langle\gamma'_\beta, \vec{a}\vec{b}\rangle$ and $\langle\gamma'_\beta, \vec{a}\vec{b}\rangle^{\checkmark}$ for some γ'_β (there must exist such a $\vec{b}$ since $AbleBy(m(\beta), \vec{a}, \gamma_\beta)$). Then by Assumption 7.4, we also have that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}\vec{b}, S_0))\} \models Do(m(\beta), do(\vec{a}, S_0), do(\vec{a}\vec{b}, S_0))$. So $AbleBy(\delta'_h, \vec{\alpha}\beta, \gamma'_h)$ and all the other conditions in the antecedent of the induction hypothesis hold and we can apply it to get that there exists γ_b such that $AbleBy(m_p(\gamma'_h), \vec{a}\vec{b}, \gamma_b)$. By Lemma A.29 we then have $AbleBy(m(\beta); m_p(\gamma'_h), \vec{a}, \mathbf{seq}(\gamma_\beta, G))$ where $G = \{\vec{b}; \gamma_b \mid$ such that $\langle\gamma_\beta, \vec{a}\rangle \rightarrow^*_{\vec{b}} \langle\gamma'_\beta, \vec{a}\vec{b}\rangle$ and $\langle\gamma'_\beta, \vec{a}\vec{b}\rangle^{\checkmark}$ for some γ'_β and $AbleBy(m_p(\gamma'_h), \vec{a}\vec{b}, \gamma_b)\}$. Thus $AbleBy(m_p(\gamma_h), \vec{a}, \mathbf{seq}(\gamma_\beta, G))$ and the thesis follows.

Case 2: $\gamma_h = \mathbf{set}(E)$ where E is a non-empty set of programs of the form $[\beta^e_i; \gamma_i]$ where $\beta^e_i \in \mathcal{A}^e$ and γ_i is a strategy for all i.
Since we have $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$, it then follows by the definition of $AbleBy$ that there exists an exogenous high-level action $\beta \in \mathcal{A}^e_h$, a high-level strategy γ'_h, and a high-level program δ'_h such that $\langle\delta_h, \vec{\alpha}\rangle \rightarrow_\beta \langle\delta'_h, \vec{\alpha}\beta\rangle$ and $AbleBy(\delta'_h, \vec{\alpha}\beta, \gamma'_h)$. The former implies that $D_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta, do(\vec{\alpha}, S_0))\}$ is satisfiable.
From this, it follows by Assumption 7.1 (Turn-Taking) and 7.3 (Always Known Whose Turn It Is) that there does not exist a high-level ordinary action $\beta' \in \mathcal{A}^o_h$ such that $D_h \cup \{Executable(do(\vec{\alpha}, S_0)) \wedge Poss(\beta', do(\vec{\alpha}, S_0))\}$ is satisfiable. Since we have a sound abstraction, it then follows by Proposition 7.3 that there does not exist a high-level ordinary action $\beta' \in \mathcal{A}^o_h$ such that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta'), do(\vec{a}, S_0), s)\}$

is satisfiable.
From this, it follows by Assumption 7.2 (Non-Blocking) that there exists an exogenous high-level action $\beta \in \mathcal{A}_h^e$ such that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta), do(\vec{a}, S_0), s)\}$ is satisfiable.
Thus by Assumption 7.6 (Exogenous HL Actions Never Block), we have that $NecTerminates(\textsc{anyoneexohl}, \vec{a})$, and furthermore by Lemma A.31, we have that there exists a low-level strategy γ_1 such that $AbleBy(\textsc{anyoneexohl}, \vec{a}, \gamma_1)$.
Now take arbitrary β_i and $\vec{b}_{i,j}$ such that $\langle m(\beta_i), \vec{a}\rangle \rightarrow^*_{\vec{b}_{i,j}} \langle \delta_l, \vec{a}\vec{b}_{i,j}\rangle$ and $\langle \delta_l, \vec{a}\vec{b}_{i,j}\rangle^{\checkmark}$ for some δ_l.
Since there does not exist a high-level ordinary action $\beta' \in \mathcal{A}_h^o$ such that $D_l \cup \mathcal{C} \cup \{Executable(do(\vec{a}, S_0)) \wedge \exists s.Do(m(\beta', do(\vec{a}, S_0), s)\}$ is satisfiable, it follows that $\beta_i \in \mathcal{A}_h^e$.
Moreover by Assumption 5.2, we have that every online execution from $do(\vec{a}, S_0)$ starts with an execution of a refinement of such a high-level action.
Since we have $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$, it follows by the definition of $AbleBy$ that there exists β_i, γ_i, such that $\langle \beta_i; \gamma_i, \vec{\alpha}\rangle \rightarrow_{\beta_i} \langle \gamma_i, \vec{\alpha}\beta_i\rangle$.
By Assumption 7.4 (Awareness of Executed HL Actions), we have that $D_h \cup \{Executable(do(\vec{\alpha}\beta_i, S_0))\}$ is a sound abstraction wrt m of $D_l \cup \{Executable(do(\vec{a}\vec{b}_{i,j}, S_0))\}$.
All the other conditions in the antecedent of the induction hypothesis hold and we can apply it to get that there exists a low-level strategy $\gamma_{i,j}$ such that $AbleBy(m(\gamma_i), \vec{a}\vec{b}_{i,j}, \gamma_{i.j})$.
Finally, by Lemma A.29, we then have $AbleBy(m_p(\gamma_h), \vec{a}, \mathbf{seq}(\gamma_1, G))$, where

$$\begin{aligned} G = \{\vec{b}_{i,j}; \gamma_{i,j} \mid\ & \text{there exists } \beta_i \text{ such that} \\ & \langle \gamma_h, \vec{\alpha}\rangle \rightarrow_{\beta_i} \langle \gamma_i, \vec{\alpha}\beta_i\rangle \\ & \text{and } \langle m(\beta_i), \vec{a}\rangle \rightarrow^*_{\vec{b}_{i,j}} \langle \delta', \vec{a}\vec{b}_{i,j}\rangle \\ & \text{and } \langle \delta', \vec{a}\vec{b}_{i,j}\rangle^{\checkmark} \text{ for some } \delta' \\ & \text{and } \langle \gamma_1, \vec{a}\rangle \rightarrow^*_{\vec{b}_{i,j}} \langle \gamma', \vec{a}\vec{b}_{i,j}\rangle \\ & \text{and } \langle \gamma', \vec{a}\vec{b}_{i,j}\rangle^{\checkmark} \text{ for some } \gamma' \\ & \text{and } AbleBy(m(\gamma_i), \vec{a}\vec{b}_{i,j}, \gamma_{i,j})\} \end{aligned}$$

and the thesis follows. □

Lemma A.32 *For any online situation-determined program δ, any ground sequence of actions $\vec{a}$, any strategy γ, and any situation suppressed formula ϕ, if $AbleBy(\delta, \vec{a}, \gamma)$ and $\mathcal{D} \cup \mathcal{C} \models Do(\delta, do(\vec{a}, S_0), s') \supset \phi[s']$, then $AbleBy(\delta; \phi?, \vec{a}, \gamma)$.*

Proof (Sketch) By induction on length of γ. □

Corollary 7.7 Suppose that $\mathcal{D}_h$ is a sound abstraction of $\mathcal{D}_l$ relative to mapping m and Assumptions 5.2, 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6 hold. Then for all ground high-level action sequences $\vec{\alpha}$ and all ground low-level action sequences $\vec{a}$ such that $\langle \vec{\alpha}, \epsilon\rangle \rightarrow^*_{\vec{\alpha}} \langle \delta'_h, \vec{\alpha}\rangle$ for some δ'_h and $\langle \delta'_h, \vec{\alpha}\rangle^{\checkmark}$ and $\langle m(\vec{\alpha}), \epsilon\rangle \rightarrow^*_{\vec{a}} \langle \delta_l, \vec{a}\rangle$ and $\langle \delta_l, \vec{a}\rangle^{\checkmark}$ for some δ_l and for any high-level online situation determined program δ_h, for any high-level strategy γ_h, and any situation-suppressed formula ϕ, if $\mathcal{D}_h \models Do(\delta_h, do(\vec{\alpha}, S_0), s') \supset \phi[s']$ and $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$ then there exists γ_l such that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$ and $AbleBy(m_p(\gamma_h); m(\phi)?, \vec{a}, \gamma_l)$.

Proof We have $AbleBy(\delta_h, \vec{\alpha}, \gamma_h)$ and $\mathcal{D}_h \models Do(\delta_h, do(\vec{\alpha}, S_0), s') \supset \phi[s']$. By Lemma A.32 this implies $\mathcal{D}_h \models Do(\gamma_h, do(\vec{\alpha}, S_0), s') \supset \phi[s']$.

By Theorem 7.6, we have that $AbleBy(m_p(\gamma_h), \vec{a}, \gamma_l)$.

Now we need to show that $\mathcal{D}_l \cup \mathcal{C} \models Do(m_p(\gamma_h), do(\vec{a}, S_0), s') \supset m(\phi)[s']$. Take some arbitrary model M_l of $\mathcal{D}_l \cup \mathcal{C}$. Since we have a sound abstraction, there exist a model M_h of $\mathcal{D}_h$ such that $M_h \sim_m M_l$. By Lemma A.19 we know that $\mathcal{CR}_{M_l}(m_p(\gamma_h), do(\vec{a}, S_0)) = \cup_{\vec{\beta} \in \mathcal{CR}_{M_h}(\gamma_h, do(\vec{\alpha}, S_0))} \mathcal{CR}_{M_l}(m(\vec{\beta}), do(\vec{a}, S_0))$. Thus by Theorem 7.2, $\mathcal{D}_l \cup \mathcal{C} \models Do(m_p(\gamma_h), do(\vec{a}, S_0), s') \supset m(\phi)[s']$. It follows by Lemma A.32 that $AbleBy(m_p(\gamma_h); m(\phi)?, \vec{a}, \gamma_l)$.

□

B Trans and Final for Mutual Exclusive Blocks

The **atomic**() construct corresponds to the *atm*() construct introduced in [41]. *Trans* and *Final* for *atm*() are defined as:

Trans and *Final* as before except for the following.

- We introduce indivisible programs $atm(\delta)$ and indivisible program in execution $atmc(\delta)$:

$$\begin{aligned}&Trans(atm(\delta), s, \delta', s') \equiv\\&\quad \exists\gamma.\delta' = atmc(\gamma) \wedge Trans(\delta, s, \gamma, s')\end{aligned}$$

$$\begin{aligned}&Trans(atmc(\delta), s, \delta', s') \equiv\\&\quad \exists\gamma.\delta' = atmc(\gamma) \wedge Trans(\delta, s, \gamma, s')\end{aligned}$$

$$Final(atm(\delta), s) \equiv Final(\delta, s)$$

$$Final(atmc(\delta), s) \equiv Final(\delta, s)$$

- We define the property $AtmOn(\delta, s)$ (by induction on δ or by fixpoint in case of procedures) to say that a subprogram in δ in s is indivisible program in execution, i.e., has the form $atmc(\gamma)$ and is not *Final*:

$$\begin{aligned}&AtmOn(\delta, s) \equiv\\&\exists\gamma.\delta = atmc(\gamma) \wedge \neg Final(atmc(\gamma), s) \vee\\&\exists\gamma_1, \gamma_2.\delta = \gamma_1; \gamma_2 \wedge AtmOn(\gamma_1, s) \vee\\&\exists\gamma_1, \gamma_2.\delta = \gamma_1 \| \gamma_2 \wedge (AtmOn(\gamma_1, s) \vee AtmOn(\gamma_2, s)) \vee\\&\exists\gamma_1, \gamma_2.\delta = \gamma_1 \rangle\!\rangle \gamma_2 \wedge (AtmOn(\gamma_1, s) \vee AtmOn(\gamma_1, s))\end{aligned}$$

  This definition reflects the fact that $atmc(\gamma)$ can be embedded only within three basic contexts, namely $atmc(\gamma); \delta_2$, $atmc(\gamma) \| \delta_2$ and $atmc(\gamma) \rangle\!\rangle \delta_2$. Indeed, ; , $\|$, and $\rangle\!\rangle$, are the only constructs that ever get inserted in the remaining program after a transition and these are the only places where there can be an *atmc* (e.g., | and **if** get consumed by their transitions; iterations don't get consumed, but are inserted in the remaining program inside a ; or $\|$).

- Then we redefine concurrency and prioterized concurrency:

$$\begin{array}{l}
\mathit{Trans}(\delta_1 \| \delta_2, s, \delta', s') \equiv \\
\exists \gamma.\delta' = \gamma \| \delta_2 \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \wedge \neg \mathit{AtmOn}(\delta_2, s) \vee \\
\exists \gamma.\delta' = \delta_1 \| \gamma \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \neg \mathit{AtmOn}(\delta_1, s)
\end{array}$$

$$\begin{array}{l}
\mathit{Trans}(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv \\
\exists \gamma.\delta' = \gamma \rangle\!\rangle \delta_2 \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \wedge \neg \mathit{AtmOn}(\delta_2, s) \vee \\
\exists \gamma.\delta' = \delta_1 \rangle\!\rangle \gamma \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \neg \mathit{AtmOn}(\delta_1, s) \wedge \\
\qquad \neg \exists \gamma, s'. \mathit{Trans}(\delta_1, s, \gamma, s')
\end{array}$$

- Note that, with this characterization of *Trans* and *Final*, one should be able to prove that at most a subprogram at the time can be an indivisible program in execution (i.e., at most one subprogram can be of the form $atmc(\gamma)$).

- Note also that this definition does not allow exogenous actions to be interleaved into the atomic block. If we want to allow this, we need to define concurrency as follows:

$$\begin{array}{l}
\mathit{Trans}(\delta_1 \| \delta_2, s, \delta', s') \equiv \\
\quad \exists \gamma.\delta' = \gamma \| \delta_2 \wedge \mathit{Trans}(\delta_1, s, \gamma, s') \wedge \\
\qquad (\neg \mathit{AtmOn}(\delta_2, s) \vee s' = do(a, s) \wedge \mathit{Exo}(a)) \vee \\
\quad \exists \gamma.\delta' = \delta_1 \| \gamma \wedge \mathit{Trans}(\delta_2, s, \gamma, s') \wedge \\
\qquad (\neg \mathit{AtmOn}(\delta_1, s) \vee s' = do(a, s) \wedge \mathit{Exo}(a))
\end{array}$$

  Similarly for prioterized concurrency.

Bibliography

[1] Javier Segovia Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. Hierarchical finite state controllers for generalized planning. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 3235–3241, 2016.

[2] Natasha Alechina, Nils Bulling, Mehdi Dastani, and Brian Logan. Practical run-time norm enforcement with bounded lookahead. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 443–451. ACM, 2015.

[3] Ronald Alford, Ugur Kuter, Dana S. Nau, Elnatan Reisner, and Robert P. Goldman. Maintaining focus: Overcoming attention deficit disorder in contingent planning. In *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*, 2009.

[4] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[5] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, 2007.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[7] Guillaume Aucher. Supervisory control theory in epistemic temporal logic. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014)*. Springer, 2014.

[8] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[10] Jorge A. Baier and Sheila A. McIlraith. On planning with programs that sense. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 492–502. AAAI Press, 2006.

[11] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Abstraction in situation calculus action theories - Extended version. Technical Report EECS-2016-04, York University, 2016.

[12] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Online agent supervision in the situation calculus. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 922–928. IJCAI/AAAI Press, 2016.

[13] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Online agent supervision in the situation calculus - Extended version. Technical Report EECS-2016-02, York University, 2016.

[14] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Online situation-determined agents and their supervision. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning*, pages 517–520. AAAI Press, 2016.

[15] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Abstraction in situation calculus action theories. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 1048–1055. AAAI Press, 2017.

[16] Bita Banihashemi, Giuseppe De Giacomo, and Yves Lespérance. Hierarchical agent supervision. Submitted to AAMAS 2018, 2018.

[17] Francesco Belardinelli, Alessio Lomuscio, and Jakub Michaliszyn. Agent-based refinement for predicate abstraction of multi-agent systems. In *Proceedings of the 22nd European Conference on Artificial Intelligence*, pages 286–294, 2016.

[18] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[19] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thruna. Decision-Theoretic, High-Level Agent Programming in the Situation Calculus. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence*, pages 355–362. AAAI Press / The MIT Press, 2000.

[20] Olivier Boutin, Jan Komenda, Tomás Masopust, Klaus Schmidt, and Jan H. van Schuppen. Hierarchical control with partial observations: Sufficient conditions. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference*, pages 1817–1822. IEEE, 2011.

[21] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[22] Michael E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[23] Kai Cai and W. Murray Wonham. Supervisor localization: A top-down approach to distributed control of discrete-event systems. *IEEE Transactions on Automatic Control*, 55(3):605–618, 2010.

[24] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008.

[25] Sheng-Luen Chung, Stéphane Lafortune, and Feng Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, 1992.

[26] A. Cimatti, Marco Pistore, M. Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[27] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[28] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[29] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 11th International Conference*, pages 589–599, 2008.

[30] Jens Claßen, Martin Liebenberg, Gerhard Lakemeyer, and Benjamin Zarrieß. Exploring the boundaries of decidable verification of non-terminating golog programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 1012–1019. AAAI Press, 2014.

[31] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming multi-agent systems in 3apl. In *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer, 2005.

[32] Ernest Davis. Knowledge preconditions for plans. *Journal of Logic and Computation*, 4(5):721–766, 1994.

[33] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[34] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. IndiGolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming: Languages, Tools and Applications*, pages 31–72. Springer US, Boston, MA, 2009.

[35] Giuseppe De Giacomo, Yves Lespérance, and Christian Muise. On supervising agents in situation-determined ConGolog. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 1031–1038, Valencia, Spain, 2012.

[36] Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi. Bounded situation calculus action theories. *Artificial Intelligence*, 237:172–203, 2016.

[37] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Sebastian Sardiña. Verifying ConGolog programs on bounded situation calculus theories. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, pages 950–9568. AAAI Press, 2016.

[38] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus-based programs for representing and reasoning about game structures. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning*, pages 445–455, Toronto, ON, Canada, 2010.

[39] Giuseppe De Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, 1999.

[40] Giuseppe De Giacomo and Hector J. Levesque. Projection using regression and sensors. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 160–165. Morgan Kaufmann, 1999.

[41] Giuseppe De Giacomo, Hector J. Levesque, and Yves Lespérance. Trans and Final for mutual exclusive blocks. Unpublished Note, 2004.

[42] Giuseppe De Giacomo, Riccardo De Masellis, and Fabio Patrizi. Composition of partially observable services exporting their behaviour. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 1063–1069. AAAI Press, 2009.

[43] Giuseppe De Giacomo and Fabio Patrizi. Automated composition of nondeterministic stateful services. In *Web Services and Formal Methods*, volume 6194 of *Lecture Notes in Computer Science*, pages 147–160. Springer Berlin Heidelberg, 2010.

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

[45] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 453–465. Morgan Kaufmann, 1998.

[46] Giuseppe De Giacomo and Sebastian Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proceedings of the 20th international joint conference on Artificial intelligence*, pages 1866–1871, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[47] Massimiliano de Leoni, Giuseppe De Giacomo, Yves Lespèrance, and Massimo Mecella. On-line adaptation of sequential mobile processes running concurrently. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1345–1352, New York, NY, USA, 2009. ACM.

[48] D. C. Dennett. *The Intentional Stance*. The MIT Press, 1987.

[49] Frank Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, 1999.

[50] Frank Dignum, Virginia Dignum, Rui Prada, and Catholijn M. Jonker. A conceptual architecture for social deliberation in multi-agent organizations. *Multiagent and Grid Systems*, 11(3):147–166, 2015.

[51] Mark d'Inverno, Michael Luck, Michael P. Georgeff, David Kinny, and Michael Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[52] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. TAL: temporal action logics language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2:273–306, 1998.

[53] E. Allen Emerson. Model checking and the Mu-calculus. In *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop 1996*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. DIMACS/AMS, 1996.

[54] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[55] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *ACM*, 33(1):151–178, 1986.

[56] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2013 edition, 1972.

[57] Kutluhan Erol, James A. Hendler, and Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[58] Paolo Felli, Nitin Yadav, and Sebastian Sardiña. Supervisory control for behavior composition. *IEEE Transactions on Automatic Control*, 62(2):986–991, 2017.

[59] Joseph J Finger. *Exploiting constraints in design synthesis*. PhD thesis, Standford University, 1987.

[60] Christian Fritz and Yolanda Gil. Towards the integration of programming by demonstration and programming by instruction using Golog. In *AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR)*. AAAI, 2010.

[61] Christian Fritz and Sheila A. McIlraith. Decision-theoretic Golog with qualitative preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 153–163, 2006.

[62] Alfredo Gabaldon. Non-markovian control in the situation calculus. In *Proceedings of the 18th National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 519–525. AAAI Press / The MIT Press, 2002.

[63] Alfredo Gabaldon. Programming hierarchical task networks in the situation calculus. In *AIPS02 Workshop on On-line Planning and Scheduling*, 2002.

[64] Alfredo Gabaldon. Compiling control knowledge into preconditions for planning in the situation calculus. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1061–1066. Morgan Kaufmann, 2003.

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

[66] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning.* Morgan & Claypool Publishers, 2013 edition, 2013.

[67] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.

[68] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. *The Belief-Desire-Intention Model of Agency*, pages 1–10. Springer Berlin Heidelberg, 1999.

[69] Michael P. Georgeff and Francois Felix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 972–978. Morgan Kaufmann Publishers Inc., 1989.

[70] Hojjat Ghaderi. *A Logical Theory of Joint Ability in the Situation Calculus.* PhD thesis, University of Toronto, 2010.

[71] Chiara Ghidini and Fausto Giunchiglia. Local models semantics, or contextual reasoning=locality+compatibility. *Artificial Intelligence*, 127(2):221–259, 2001.

[72] Chiara Ghidini and Fausto Giunchiglia. A semantics for abstraction. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 343–347. IOS Press, 2004.

[73] Yolanda Gil, Varun Ratnakar, and Christian Fritz. Assisting Scientists with Complex Data Analysis Tasks through Semantic Workflows. In *Proactive Assistant Agents, Papers from the 2010 AAAI Fall Symposium*, volume FS-10-07 of *AAAI Technical Report*, pages 3101–3106. AAAI, 2010.

[74] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 5(2):323–389, 1992.

[75] Alexandra Goultiaeva and Yves Lespérance. Incremental plan recognition in an agent programming framework. In *Working Notes of the AAAI 2007 Workshop on Plan, Activity, and Intent Recognition (PAIR'07)*. 2007.

[76] Lenko Grigorov and Karen Rudie. Near-optimal online control of dynamic discrete-event systems. *Discrete Event Dynamic Systems*, 16(4):419–449, 2006.

[77] Lenko Grigorov and Karen Rudie. Dynamic discrete-event systems with instances for the modeling of emergency response protocols. In *American Control Conference (ACC)*, pages 4478–4483, 2011.

[78] Davide Grossi and Frank Dignum. Abstract and concrete norms in institutions. Technical Report UU-CS-2004-016, Utrecht University, 2004.

[79] Davide Grossi and Frank Dignum. From abstract to concrete norms in agent institutions. In *Proceedings of the 3rd International Workshop on Formal Approaches to Agent-Based Systems*, volume 3228, pages 12–29. Springer, 2004.

[80] Yilan Gu and Mikhail Soutchanski. Decidable reasoning in a modified situation calculus. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1891–1897, 2007.

[81] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic.* MIT Press, Cambridge, MA, USA, 2000.

[82] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 453–462, 1995.

[83] Jaakko Hintikka. *Knowledge and Belief.* Cornell University Press, 1922 edition, 1962.

[84] Wilfrid Hodges. *A Shorter Model Theory.* Cambridge University Press, New York, NY, USA, 1997.

[85] Yuxiao Hu and Hector J. Levesque. Planning with loops: Some new results. In *ICAPS Workshop on Generalized Planning: Macros, Loops, Domain Control.*, 2009.

[86] Yuxiao Hu and Hector J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2638–2643. IJCAI/AAAI, 2011.

[87] Kemal Inan. Nondeterministic supervision under partial observations. In *Proceedings of the 11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, volume 199 of *Lecture Notes in Control and Information Sciences*, pages 39–48. Springer Berlin Heidelberg, 1994.

[88] Kemal M. Inan and Pravin P. Varaiya. Algebras of Discrete Event Models. *Proceedings of the IEEE*, 77(1):24–38, 1989.

[89] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[90] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[91] Saul A. Kripke. Semantical analysis of modal logic i. normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, (9):67–96, 1963.

[92] Ratnesh Kumar and Mark A. Shayman. Centralized and decentralized supervisory control of nondeterministic systems under partial observation. *SIAM Journal on Control and Optimization*, 35(2):363–383, 1997.

[93] Ugur Kuter, Dana Nau, Elnatan Reisner, and Robert Goldman. Conditionalization: Adapting forward-chaining planners to partially observable environments. In *ICAPS 2007 - Workshop on planning and execution for real-world systems*, 2007.

[94] Jonas Kvarnström and Patrick Doherty. Talplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169, 2000.

[95] Alexander Lazovik and Heiko Ludwig. Managing process customizability and customization: Model, language and process. In *Web Information Systems Engineering - WISE 2007*, volume 4831 of *Lecture Notes in Computer Science*, pages 373–384. Springer Berlin Heidelberg, 2007.

[96] Ryan J. Leduc, Mark Lawford, and W. Murray Wonham. Hierarchical interface-based supervisory control-part ii: Parallel case. *IEEE Transactions on Automatic Control*, 50(9):1336–1348, 2005.

[97] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.

[98] Yves Lespérance. On the epistemic feasibility of plans in multiagent systems specification. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001*, volume 2333 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2002.

[99] Yves Lespérance, Giuseppe De Giacomo, and Atalay Nafi Ozgovde. A model of contingent planning for agent programming languages. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS*, pages 477–484. IFAAMAS, 2008.

[100] Yves Lespérance, Hector J. Levesque, Fangzhen Lin, and Richard B. Scherl. Ability and knowing how in the situation calculus. *Studia Logica*, 66(1):165–186, 2000.

[101] Hector J. Levesque. What is planning in the presence of sensing? In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference*, pages 1139–1146, 1996.

[102] Hector J. Levesque and Gerhard Lakemeyer. *The logic of knowledge bases.* MIT Press, 2000.

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

[104] Qianhui Liang, Xindong Wu, E.K. Park, T.M. Khoshgoftaar, and Chi hung Chi. Ontology-based business process customization for composite web services. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 41(4):717–729, 2011.

[105] Sotirios Liaskos, Shakil M. Khan, Marin Litoiu, Marina Daoud Jungblut, Vyacheslav Rogozhkin, and John Mylopoulos. Behavioral adaptation of information systems through goal models. *Information Systems*, 37(8):767–783, 2012.

[106] Sotirios Liaskos, Marin Litoiu, Marina Daoud Jungblut, and John Mylopoulos. Goal-based behavioral customization of information systems. In *Advanced Information Systems Engineering*, volume 6741 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin Heidelberg, 2011.

[107] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[108] Alessio Lomuscio and Marek J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[109] Erick Martinez and Yves Lespérance. Web service composition as a planning task: Experiments using knowledge-based planning. In *Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services*, pages 62–69, June 2004.

[110] John McCarthy and Patrick J. Hayes. Some Philosophical Problems From the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[111] Sheila A. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 167–179. Morgan Kaufmann, 1998.

[112] Sheila A. McIlraith and Ronald Fadel. Planning with complex actions. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 356–364, 2002.

[113] Sheila A. McIlraith and Tran Cao Son. Adapting Golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning*, pages 482–493, 2002.

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

[115] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[116] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[117] Robin Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.

[118] Peiming Mo, Naiqi Li, and Yongmei Liu. Automatic verification of Golog programs via predicate abstraction. In *Proceedings of the 22nd European Conference on Artificial Intelligence*, pages 760–768, 2016.

[119] Robert C. Moore. A formal theory of knowledge and action. In *Formal Theories of the Commonsense World*, pages 319–358. Ablex Publishing Corporation, Norwood, New Jersey, 1985.

[120] Dana Nau, Malik Ghallab, and Paolo Traverso, editors. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[121] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[122] P. Pandurang Nayak and Alon Y. Levy. A semantic theory of abstractions. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 196–203, 1995.

[123] Quang Ha Ngo and Kiam Tian Seow. Command and control of discrete-event systems: Towards online hierarchical control based on feasible system decomposition. *IEEE Transactions on Automation Science and Engineering*, 11(4):1218–1228, 2014.

[124] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[125] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[126] Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference*, pages 613–622. AAAI Press, 2004.

[127] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16(1):47–108, 1981.

[128] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[129] Peter J. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[130] Peter J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[131] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

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

[133] Laurie Ricker and Benoît Caillaud. Mind the gap: Expanding communication options in decentralized discrete-event control. *Automatica*, 47(11):2364–2372, 2011.

[134] Karen Rudie and W. Murray Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[135] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[136] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Acm transactions on programming languages and systems. *Artificial Intelligence and Law*, 24(3):217–298, 2002.

[137] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in Artificial Intelligence and Complex Systems.* Springer-Verlag New York, 2013.

[138] Sebastian Sardina and Giuseppe De Giacomo. Composition of ConGolog programs. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 904–910, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[139] Sebastian Sardiña, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On ability to autonomously execute agent programs with sensing. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1522–1523. IEEE Computer Society, 2004.

[140] Sebastian Sardiña, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in IndiGolog - from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):259–299, 2004.

[141] Sebastian Sardiña, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the limits of planning over belief states under strict uncertainty. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 463–471. AAAI Press, 2006.

[142] Sebastian Sardina and Yves Lespérance. Golog speaks the BDI language. In *Programming Multi-Agent Systems, 7th International Workshop, ProMAS 2009*, pages 82–99. Springer, 2009.

[143] Sebastian Sardiña and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.

[144] Sebastian Sardina, Fabio Patrizi, and Giuseppe De Giacomo. Behavior composition in the presence of failure. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 11th International Conference*, pages 640–650. AAAI Press, 2008.

[145] Richard B. Scherl and Hector J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1-2):1–39, 2003.

[146] K. Schmidt and E. G. Schmidt. Communication of distributed discrete-event supervisors on a switched network. In *Proceedings of the 9th International Workshop on Discrete Event Systems*, pages 419–424, 2008.

[147] Klaus Schmidt and Christian Breindl. Maximally permissive hierarchical control of decentralized discrete event systems. *IEEE Transactions on Automatic Control*, 56(4):723–737, 2011.

[148] Kiam Tian Seow. Organizational control of discrete-event systems: A hierarchical multiworld supervisor design. *IEEE Transactions on Control Systems Technology*, 22(1):23–33, 2014.

[149] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.

[150] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceeding of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 19–26. ACM, 2002.

[151] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[152] Munindar P. Singh. A logic of situated know-how. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 343–348. AAAI Press / The MIT Press, 1991.

[153] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[154] Samaneh Soltani, Mohsen Asadi, Marek Hatala, Dragan Gasevic, and Ebrahim Bagheri. Automated planning for feature model configuration based on stakeholders' business concerns. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 536–539, Washington, DC, USA, 2011. IEEE Computer Society.

[155] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Abstract planning with unknown object quantities and properties. In *Proceedings of the 8th Symposium on Abstraction, Reformulation, and Approximation*. AAAI, 2009.

[156] Josh D. Tenenberg. Preserving consistency across abstraction mappings. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 1011–1014, 1987.

[157] Michael Thielscher. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1-2):277–299, 1999.

[158] Michael Thielscher. Representing the knowledge of a robot. In *Principles of Knowledge Representation and Reasoning, Proceedings of the 7th International Conference*, pages 109–120. Morgan Kaufmann, 2000.

[159] Athanasios Tsalatsanis, Ali Yalcin, and Kimon P. Valavanis. Dynamic task allocation in cooperative robot teams. *Robatica*, 1:1–10, 2012.

[160] Wiebe van der Hoek, Bernd van Linder, and John-Jules Ch. Meyer. A logic of capabilities. In *Logical Foundations of Computer Science, 3rd International Symposium*, volume 813, pages 366–378. Springer, 1994.

[161] Ron van der Meyden and Ka shu Wong. Complete axiomatizations for reasoning about knowledge and branching time. *Studia Logica*, 75(1):93–123, 2003.

[162] Javier Vázquez-Salceda and Frank Dignum. Modelling electronic organizations. In *Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems*, volume 2691 of *Lecture Notes in Computer Science*, pages 584–593. Springer, 2003.

[163] Michael Winikoff. *Jack^{TM} Intelligent Agents: An Industrial Strength Platform*, pages 175–193. Springer US, 2005.

[164] Frank Wolter and Michael Wooldridge. Temporal and dynamic logic. *Journal of Indian Council of Philosophical Research*, XXVII(1):249–276, 2011.

[165] K. C. Wong and W. Murray Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems*, 6(3):783–800, 1996.

[166] W. Murray Wonham. *Supervisory Control of Discrete-Event Systems*. University of Toronto, 2017.

[167] Michael Wooldridge and Nicholas R. Jennings. *Agent theories, architectures, and languages: A survey*, pages 1–39. Springer Berlin Heidelberg, 1995.

[168] Liping Xiong and Yongmei Liu. Strategy representation and reasoning for incomplete information concurrent games in the situation calculus. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1322–1329. IJCAI/AAAI Press, 2016.

[169] Nitin Yadav, Paolo Felli, Giuseppe De Giacomo, and Sebastian Sardina. Supremal realizability of behaviors with uncontrollable exogenous events. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, IJCAI'13, pages 1176–1182. AAAI Press, 2013.

[170] Qiang Yang. *Intelligent planning - a decomposition and abstraction based approach*. Artificial intelligence. Springer, 1997.

[171] Renyuan Zhang, Kai Cai, Yongmei Gan, and W. Murray Wonham. Distributed supervisory control of discrete-event systems with communication delay. *Discrete Event Dynamic Systems*, 26(2):263–293, 2016.

[172] Hao Zhong. *Hierarchical Control of Discrete Event Systems*. PhD thesis, University of Toronto, 1992.

[173] Hao Zhong and W. Murray Wonham. On consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, 1990.

[174] Changyan Zhou, Ratnesh Kumar, and Shengbing Jiang. Control of nondeterministic discrete-event systems for bisimulation equivalence. *IEEE Transactions on Automatic Control*, 51(5):754–765, 2006.