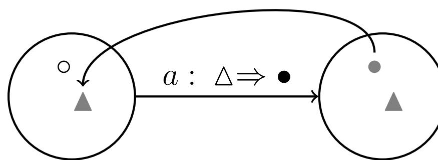
POSTDICITVE REASONING IN EPISTEMIC ACTION THEORY

MANFRED EPPE



DISSERTATION

Vorgelegt im Fachbereich 3
(Mathematik und Informatik)
der
Universität Bremen
zur Erlangung des Grades

DOKTOR DER INGENIEURWISSENSCHAFTEN
– Dr.-Ing. –

Bremen, 18. November 2013

Datum des Promotionskolloquiums: 18. Dezember 2013

Gutachter:

Prof. Dr. Bernd Krieg-Brückner (Universität Bremen, Deutschland)

Prof. Joohyung Lee, PhD (Arizona State University, USA)

Abstract

If an agent executes an action, this will not only change the world physically, but also the agent's knowledge about the world. Therefore the occurrence of an action can be modeled as an epistemic state transition, which maps the knowledge state of an agent to a successor knowledge state. For example, consider that an agent in a state s_0 executes an action a. This causes a transition to a state s_1. Subsequently, the agent executes a sensing action a_s, which produces knowledge and causes a transition to a state s_2. With the information gained by the sensation, the agent can not only extend its knowledge about s_2, but also infer additional knowledge about the initial state s_0. That is, the agent uses knowledge about the present to retrospectively acquire additional information about the past. We refer to this temporal form of epistemic inference as *postdiction*.

$$\overset{\text{Postdiction}}{\overleftarrow{s_0 \xrightarrow{a} s_1 \xrightarrow{a_s} s_2}}$$

Existing action theories are not capable of efficiently performing postdictive reasoning because they require an exponential number of state variables to represent an agent's knowledge state.

The contribution of this thesis is an approximate epistemic action theory, which is capable of postdictive reasoning, while it requires only a linear number of state variables to represent an agent's knowledge state. In addition, the theory is able to perform a more general *temporal* form of postdiction, which most existing approaches do not support. We call the theory the *h-approximation* – $\mathcal{HPX}$– because it explicitly represents "historical" knowledge about past world states.

In addition to the operational semantics of $\mathcal{HPX}$, we present its formalization in terms of Answer Set Programming (ASP) and provide respective soundness results. The ASP implementation allows us to apply $\mathcal{HPX}$ in real robotic applications by using off-the-shelf ASP solvers. Specifically, we integrate of $\mathcal{HPX}$ in an online planning framework for Cognitive Robotics where planning, plan execution and abductive explanation tasks are interleaved.

As a proof-of-concept, we provide a case study that demonstrates the application of $\mathcal{HPX}$ for high-level robot control in a smart home. The case-study emphasizes the usefulness of postdiction for *abnormality detection* in robotics: actions performed by robots are often not successful due to unforeseen practical problems. A solution is to verify action success by observing the effects of the action. If the desired effects do not hold after action execution, then one can postdict the existence of an abnormality and perform failure diagnosis.

Acknowledgements

This writing of this dissertation would not have been possible without the generous help from many colleagues and friends. First of all I would like to thank Bernd Krieg-Brückner. Bernd, it was your persistent and endurant support which made it possible for me to write this thesis. You kept supporting me, especially during hard times. Thank You!

I also want to thank Joohyung Lee for reviewing the thesis and for his support and feedback when we discussed my scientific contributions.

I am also indebted to Mehul Bhatt. Mehul, you kept me motivated during the last years and you taught me how to structure scientific thoughts. Your constant support and advice was substantial for the writing of this thesis. I am going to miss our fruitful nightly discussions in Rotkäppchen and the Indian dinners a lot.

I also thank Dieter Hutter. Dieter, you spent a lot of time and energy on teaching me how to approach mathematical problems, and I really learned a lot from this. Thank you!

Frank Dylla, I am indebted to you for the fruitful collaborations and for your passionate support when I developed the core contributions of this thesis.

I thank Christian Freksa who provided me the infrastructure of his group; I thank Christian Mandel and Bernd Gersdorf who were always happy to help when it came to the practical application in the BAALL laboratory, and I thank Oliver Kutz who always had an answer to questions about logic. Many other people supported me in the writing of this thesis in one or another way and it is impossible to name all of them. I hereby want to thank everybody who supported me in the last three years with all my heart.

Contents

1. Introduction 1

1.1. Reasoning about Action, Change and Knowledge 2

1.1.1. Epistemic and Non-epistemic State Transitions 3

1.1.2. Reasoning About Possible Worlds 5

1.1.3. Approximations of the Possible Worlds Semantics 5

1.2. Contributions of this Thesis 9

1.3. Thesis Outline 14

2. Background and Related Work 17

2.1. Reasoning about Action, Change and Knowledge 17

2.1.1. Operational Semantics 18

2.1.2. Model-theoretic Semantics 21

2.1.3. Non-monotonicity and Circumscription 23

2.2. Answer Set Programming 24

2.2.1. Positive Logic Programs 25

2.2.2. Grounding 26

2.2.3. Normal Logic Programs and Negation as Failure 27

2.2.4. Extensions to the Stable Model Semantics 29

2.2.5. Computational Properties 33

2.2.6. ASP-based Action Theory and Negation as Failure 33

2.2.7. ASP Module Theory 34

2.2.8. Iterative ASP Solving 35

2.2.9. Incremental Online ASP Solving 38

2.3. Modal Logic and the Possible Worlds Model of Knowledge 39

2.4. Epistemic Action Theory: A Survey 41
2.4.1. $\mathcal{PWS}$-based Epistemic Action Theories 44
2.4.2. Theories with a Disjunctive Knowledge State Representation . . 47
2.4.3. Approximate Epistemic Action Theories 47
2.4.4. Epistemic Action Theories with Explicit Knowledge-Level Effects 48

3. $\mathcal{HPX}$: The h-Approximation 51
3.1. Domain Specification and Syntax 51
3.2. Operational Semantics of the H-Approximation 53
3.2.1. Knowledge States with a Temporal Dimension 53
3.2.2. Initial Knowledge 54
3.2.3. Knowledge about the Presence (and the Past) 55
3.2.4. Executability of Actions 56
3.2.5. Sensing, Branching, Transition Function 56
3.2.6. Intermediate h-states 57
3.2.7. Inference Mechanisms (**IM1.–IM.5**) 57
3.2.8. Re-evaluation of Knowledge-level Effects 61
3.2.9. Concurrent Conditional Plans 63
3.2.10. Extended Transition Function 65
3.2.11. Plan Verification – Weak and Strong Goals 65
3.3. Computational Complexity of $\mathcal{HPX}$ 66
3.4. Relation Between $\mathcal{HPX}$ and $\mathcal{PWS}$: A Temporal Semantics for the Action Language $\mathcal{A}_k$ 66
3.4.1. Syntactical Mapping Between $\mathcal{A}_k$ and $\mathcal{HPX}$ 67
3.4.2. Original $\mathcal{PWS}$-based $\mathcal{A}_k$ Semantics 67
3.4.3. Temporal Query Semantics – ${\mathcal{A}_k}^{TQS}$ 70
3.4.4. Relation between $\mathcal{A}_k$ and $\mathcal{A}_k^{TQS}$ 72
3.4.5. Soundness of $\mathcal{HPX}$ wrt. ${\mathcal{A}_k}^{TQS}$ 72

4. Answer Set Programming Formalization of $\mathcal{HPX}$ 73
4.1. Main Predicates and Notation 73
4.2. Constitution of an $\mathcal{HPX}$-Logic Program 75
4.3. Translation Rules: ($\mathcal{D} \overset{\textbf{T1–T8}}{\longmapsto} \Gamma_{world}$) 75
4.4. Γ_{hpx} – Foundational Theory (F1) – (F7) 80
4.5. Plan Extraction from Stable Models 84
4.6. Relation between the ASP Implementation and the Operational $\mathcal{HPX}$-Semantics 85

5. An $\mathcal{HPX}$ Online Planning Framework 89
5.1. System Architecture 89

5.2. Extensions for Online Planning . . . 91
5.2.1. Incremental Planning Horizon Extension . . . 91
5.2.2. Interleaving Planning and Plan Execution . . . 91
5.2.3. Exogenous Events and Abductive Explanation . . . 94
5.2.4. Performance Optimization: Static Relations and Impossible Actions . . . 96
5.3. Incremental Reactive Planning . . . 97
5.4. Extending Expressiveness: Typing . . . 98

6. Application in a Smart Home and Evaluation 103
6.1. Case Study 1: Abnormality Detection in a Smart Home . . . 103
6.1.1. Trace: Conditional Planning with Abnormalitiy Postdiction . . . 104
6.1.2. Results and Discussion . . . 107
6.2. Case Study: Interleaving Action Planning, Abnormality Detection and Abductive Explanation in a Smart Home . . . 107
6.2.1. Trace: Interplay Between Controller and ASP Solver . . . 107
6.2.2. Results and Discussion . . . 108
6.3. Empirical Comparison with other Planners . . . 111
6.3.1. Benchmark Problems . . . 111
6.3.2. Setup . . . 112
6.3.3. Discussion . . . 112

7. Discussion and Future Work 115
7.1. Discussion . . . 115
7.2. Future Work . . . 121
7.3. Summary . . . 127

A. Soundness of ASP Implementation wrt. $\mathcal{HPX}$ Semantics 129
A.1. Notational Conventions . . . 129
A.2. Proof Overview and Structure . . . 130
A.3. Soundness of Knowledge Atoms . . . 134
A.4. Application of Effect Propositions . . . 159
A.5. Sensing Results . . . 166
A.6. Auxiliary Lemmata . . . 169

B. Computational Properties of $\mathcal{HPX}$ 171
B.1. Computational Complexity . . . 171
B.2. Knowledge-persistence and Monotonicity of Re-evaluation . . . 175
B.3. Knowledge Producing Mechanisms . . . 176

C. Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$ 179
C.1. Base Step: Initial Knowledge 180
C.2. Induction Step: Knowledge Gain for Single State Transitions 181
C.3. Additional Lemmata 197

D. Source Code and Examples 203
D.1. Foundational Theory of the Offline ASP Formalization of $\mathcal{HPX}$ 203
D.2. Basic Postdiction Example: Driving Through a Door 205
D.3. Modifications For the Incremental Online ASP Implementation 207
D.3.1. The Foundational Theory for Incremental Online Planning . . . 207
D.3.2. Incremental Modularity of $\mathcal{HPX}$-Logic Programs 210
D.4. Problem Specification for Online Planning Use Case 213

E. Dissertation-related Publications 217
E.1. Publications with Shared Content 217
E.2. Other publications 218

List of Figures 219

List of Tables 221

List of Theorems and Lemmata 223

List of Definitions 225

List of Symbols 227

Bibliography 233

1 Introduction

The term *Epistemology* was coined by the Scottish philosopher James Frederick Ferrier (1854, p.49) in his book *"Institutes of Metaphysics: The Theory of Knowing and Being"*. According to Ferrier, epistemology is the *theory of knowing*. Philosophers commonly distinguish knowledge into "knowing how" and "knowing that" (e.g. (Bengson and Moffett, 2012)): the first kind of knowledge concerns procedural knowledge about *how to do* something, e.g. how to ride a bike. The second kind refers to propositional knowledge about *how things are*, e.g. that the bike is green in color.

This thesis concerns the interplay of both branches and investigates epistemology from an action-theoretic Artificial Intelligence point of view: here, *epistemic reasoning* is referred to as the logical inference about what an agent knows according to which events occur, which event occurrences the agent is aware of, and what the agent knew initially.

The agent can acquire knowledge directly through sensing, by means of communication, or it can acquire knowledge in a less direct manner, by means of deductive and abductive inference. For instance, consider a robotic agent that tries to drive blindfolded through a door from a room A into a room B. It can infer that the door is open if it knows that it was in room A before the start of the movement and that it is in room B afterwards – If the door was closed it would be stuck in front of the door and could not have reached room B.

Taking a closer look at this example, one can actually be more precise: consider that knowledge about the robot's location is acquired through a location sensor. That is, the robot senses its location, executes the "move"-command, and later senses its location again. Then, at the time the agent acquires knowledge about being in room B, it can infer that the door must have been open *at the time it was passing it*. In other words, the agent generates knowledge about how the world was at a previous point in time, but is not able to perform this inference until a later time point where additional knowledge has been acquired. In this thesis, the temporal inference of knowledge about the past by evaluating knowledge about the presence is referred to as *postdiction*. Postdiction

typically generates knowledge about the *condition* of an action (the door being open) by observing its *effect* (the robot successfully passed the door).
Existing epistemic action theories are not capable of efficiently performing postdictive inference. Two major disadvantages in present approaches are:

1. Existing approaches are computationally complex in that they require an exponential number of state-variables to represent the knowledge of an agent. The combinatorial explosion of state variables makes the practical application of these approaches intractable.

2. Existing approaches do not consider temporal aspects of knowledge. Even though they can postdict knowledge about the condition of an action by observing its effect, existing approaches are *non-temporal* in the sense that they do not say anything about the *time* at which the condition did or did not hold. We show that such *non-temporal postdiction* causes problems with actions that involve concurrent acting and sensing. This is not the case for *temporal postdiction*, where knowledge about the past is explicitly modeled.

This thesis addresses these problems and answers the following research question:

How is it possible to realize temporal postdictive reasoning *whilst avoiding a combinatorial explosion of state variables?*

The core contribution of this thesis is an epistemic action theory that is based on explicit but approximate knowledge about the past. The advantage is that the number of knowledge-state variables is linear, as compared to an exponential number for existing approaches. This results in a lower computational complexity for reasoning tasks such as action planning: the proof of Theorem 3.1 shows that the plan existence problem for our theory is in NP, while e.g. for the epistemic action language $\mathcal{A}_k$ it is in Σ_2^P (Baral et al., 2000). Despite the lower computational complexity, our theory is more expressive in the sense that knowledge about the past is explicitly represented and postdiction is temporal.

1.1. Reasoning about Action, Change and Knowledge

The research field of reasoning about action, change and knowledge deals with the inference about what an agent knows according to what it knew initially and which actions occurred. Mathematically, the occurrence of an action is understood as a state transition, where a state s is determined by a set of domain variables. Domain variables usually change their value when state transitions occur and are therefore called *fluents* (Sandewall, 1994). Fluents paired with a value are called domain *literals*. If an agent executes an action, the fluent values change; according to a transition function (denoted ϕ), which maps a set of domain literals and an action to a set of domain literals.

1.1.1. Epistemic and Non-epistemic State Transitions

In the non-epistemic case an agent has complete knowledge about the world. For instance, consider a domain with two boolean variables denoted by the symbols $\circ$ and $\triangle$. The variables' values are denoted by coloring. For example let $\circ$ / $\bullet$ denote that a robot is in room A / room B and let $\triangle$ / $\blacktriangle$ denote an open / closed door that connects the rooms. Assume that the robot can perform a blindfold "move" action a, which has the conditional effect that $\circ$ (in room A) becomes $\bullet$ (in room B) if $\triangle$ (door open) holds. We denote this by $a : \triangle \Rightarrow \bullet$.
The execution of this action is modeled as the application of a transition function ϕ as demonstrated in Figure 1.1: the state transition $\phi(a, \{\circ, \triangle\}) = \{\bullet, \triangle\}$ describes that action a was executed in state $\{\circ, \triangle\}$, resulting in a different state $\{\bullet, \triangle\}$.

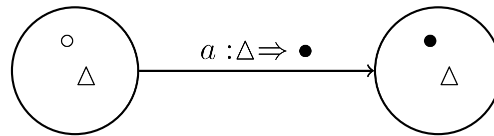


Figure 1.1.: State transition in the case of complete knowledge

This simple deduction task becomes non-trivial if we consider agents with *incomplete knowledge* about the world. One way to model incomplete knowledge is to introduce a third possible variable value *unknown* (denoted by coloring *gray*). This allows one to model three important epistemic phenomena: *knowledge loss*, *sensing* and *postdiction*.

- *Knowledge loss* is depicted in Figure 1.2: a state transition $\phi(a, \{\circ, {\color{gray}\blacktriangle}\}) = \{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}$ denotes that action a was executed in a partially unknown state $\{\circ, {\color{gray}\blacktriangle}\}$. Since it is unknown whether or not the condition of action a holds (${\color{gray}\bullet}$), one is unable to tell whether or not the effect of the action was achieved. For example, consider that the robot knows that it is in room A (denoted by $\circ$) and it does not know whether the door is open (denoted by ${\color{gray}\blacktriangle}$). If it then tries to move through the door blindfolded it will lose knowledge about its location because it can not deduce the effect of the action. Consequently, the result of the transition is a state $\{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}$ where all variable values are unknown, i.e. knowledge about $\circ$ / $\bullet$ is lost.

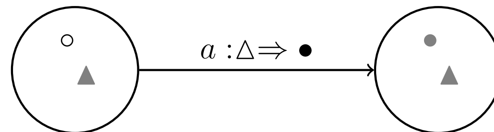


Figure 1.2.: Epistemic state transitions with knowledge loss

- *Sensing actions* can be used to acquire new knowledge. For example, let $a_s^{\bullet}$ denote a sensing action revealing that $\bullet$ holds. We model the application of this action in a state $\{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}$ as $\phi(a_s^{\bullet}, \{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}) = \{\bullet, {\color{gray}\blacktriangle}\}$ (see Figure1.3).

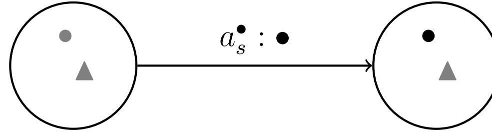


Figure 1.3.: Epistemic state transitions with sensing

- *Postdiction* is the inference of determining the condition of an action by observing its effect. For example, consider Figure 1.4, which illustrates two successive state transitions:
  1. act: $\phi(a, \{\circ, {\color{gray}\blacktriangle}\}) = \{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}$
  2. sense: $\phi(a_s^{\bullet}, \{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}) = \{\bullet, {\color{gray}\blacktriangle}\}$

  After the second state transition the sensing action $a_s^{\bullet}$ reveals that $\bullet$ holds. In this case an agent should be able to evaluate the sensing result and to *postdict* that $\triangle$ held before executing a.[1]

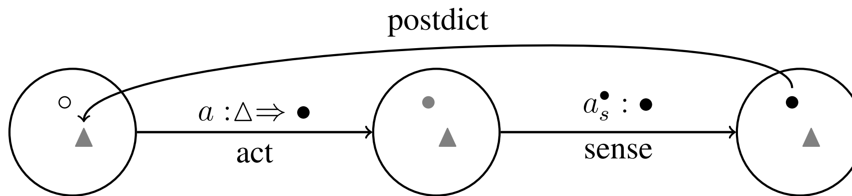


Figure 1.4.: Epistemic state transitions with postdiction

An important epistemic phenomenon, which has so far been ignored by most existing epistemic action theories, is that state transitions do not only have one but *two* temporal dimensions if considering incomplete knowledge: an "outer" dimension reflects the temporal progression of knowledge states of an agent. For example sensing that $\bullet$ holds in state $\{{\color{gray}\bullet}, {\color{gray}\blacktriangle}\}$ causes a transition towards a state $\{\bullet, {\color{gray}\blacktriangle}\}$. Within this outer dimension there is an "inner" dimension of knowledge, that enables one to express propositions like "after the agent executes $a_s^{\bullet}$ it knows that $\triangle$ held before executing a".
It is this inner dimension which our theory exploits to achieve two advantages compared to traditional possible-worlds approaches: an increase of expressiveness in terms of temporal aspects of knowledge, and a decrease of the computational complexity for solving epistemic planning problems.

[1] Under the assumption that a is the only action that happens.

1.1.2. Reasoning About Possible Worlds

The most popular approach to model an agent's knowledge is based on the Possible Worlds Semantics ($\mathcal{PWS}$). $\mathcal{PWS}$ can be traced back to Hintikka (1962), Kripke (1963) and others, who together invented *Modal Logic* (e.g. (Blackburn et al., 2001)). Years later, Moore (1985) applied the possible-worlds-approach in action theory and described dynamic epistemic systems based on an early formulation of the Situation Calculus (McCarthy, 1963).[2]
To represent knowledge and to describe adherent epistemic phenomena $\mathcal{PWS}$ does not use an extra variable value *unknown*. Its knowledge model is more precise in that it considers all possible combinations of unknown world properties separately. Each possible combination is called a *possible world*. If something holds in all possible worlds, then the agent *knows* that it holds.
The execution of physical (non-sensing) actions is modeled by separately applying the corresponding state transition to each individual possible world. Sensing is modeled as a *filter* that rules out those possible worlds, in which a sensing result does not hold. For an illustration, consider Example 1.1.
Using $\mathcal{PWS}$-based formalizations to cope with incomplete knowledge and sensing such that postdiction is supported, results in a high computational complexity: let $|\mathcal{F}|$ be the number of domain fluents. Given that all fluents are unknown, $\mathcal{PWS}$ based approaches compile one incomplete world to $2^{|\mathcal{F}|}$ complete possible worlds. Each of these worlds again contains $|\mathcal{F}|$ fluents, and modeling the knowledge state of an agent requires a total of $2^{|\mathcal{F}|} \cdot |\mathcal{F}|$ variables.
The high computational complexity that emerges from the exponential blowup[3] is a problem for applications in areas like Cognitive Robotics or Ambient Intelligence, where real-time response is needed. Though there are indeed many efficient $\mathcal{PWS}$ based planners available (e.g. ContingentFF (Hoffmann and Brafman, 2005) or MBP (Bertoli et al., 2001)), these only work well for small to mid-size problem domains. There is a phase transition, if the domain size exceeds a certain threshold.[4] If this threshold is exceeded, then the reaction time of a $\mathcal{PWS}$-driven system is not acceptable anymore.

1.1.3. Approximations of the Possible Worlds Semantics

To reduce the computational complexity of epistemic state transition systems, approximations of the $\mathcal{PWS}$ have been proposed. A prominent example is the 0-approximation for

[2]For a detailed survey on the history of possible worlds based semantics we refer to (Goldblatt, 2003).

[3]Baral et al. (2000) show that if that the plan length is polynomial and the number of sensing actions is limited then the plan-existence problem for the action language $\mathcal{A}_k$ is Σ_2^P complete.

[4]For example, the RING problem (e.g. (Hoffmann and Brafman, 2005)) where an agent must move through n rooms and close the windows in these rooms demands 1.5s for 4 rooms, 480s for 5 rooms and produces a timeout > 3600s for 6 rooms with the ContingentFF planner on a 2 Ghz i5 machine with 6GB RAM. See Table 6.2 for details.

Example 1.1 Possible worlds model of knowledge
Consider Figure 1.5: the considered domain (i.e. the world) has two boolean state variables, △ / ▲ and ○ / ●. In the initial state $[S_0]$ it is unknown whether △ / ▲, respectively ○ / ● holds. This lack of knowledge is represented by the explicit consideration of all possible combinations of variable-value pairs: $s_0^a = \{\circ, \blacktriangle\}$, $s_0^b = \{\bullet, \triangle\}$, $s_0^c = \{\bullet, \blacktriangle\}$ and $s_0^d = \{\circ, \triangle\}$.We call s_0^a – s_0^d *possible worlds* that constitute the knowledge state of an agent.
The agent can execute an action a that causes ● to hold under the condition that △ holds (denoted as $a :\triangle\Rightarrow \bullet$). After the execution of this action there are four new possible worlds: $s_1^a = \{\circ, \blacktriangle\}$, $s_1^b = \{\bullet, \triangle\}$, $s_1^c = \{\bullet, \blacktriangle\}$ and $s_1^d = \{\bullet, \triangle\}$.[a] In s_1^a – s_1^c nothing has changed, because (i) the condition △ of action a does not hold (s_0^a and s_0^c) or (ii) the effect did already hold in the initial state (s_0^b and s_0^c). In s_1^d the world changed in that ○ became ●.
Consider a sensing action a_s that determines whether ○ or ● holds. If an agent is planning to execute this action, then it has to consider two possible outcomes of the sensing actions and create two possible future *branches*, one for each possible sensing result. Sensing is modeled as a "filter", which assigns those possible worlds that coincide with a potential sensing result to one branch. In the example, two branches are created.

(S_2^a) The first branch reflects that ○ is the sensing result (denoted $a_s^\circ : \circ$). Since only state s_1^a coincides with this sensing result, only s_1^a is assigned to the branch. In this branch, all remaining possible worlds agree on both variables, i.e. in all worlds which passed the sensing-filter, ○ and ▲ hold. Since both ○ and ▲ hold in *all worlds which are possible* after the execution of a and under the assumption that ○ will be the sensing outcome, we say that ○ and ▲ are *known to hold*.

The implicit postdictive inference which is made in this case is that knowledge about △ was generated even though only ○ was sensed. That is, by ruling out those possible worlds that do not coincide with the sensing result, additional knowledge is *postdicted*.

(S_2^{b-d}) The second branch reflects that ● is the sensing result (denoted as $a_s^\bullet : \bullet$). States s_1^b – s_1^d coincide with this sensing result and are assigned to one branch. In this case, all remaining possible worlds agree on ●, and hence the agent knows that ● holds. However, the possible worlds do not agree on △ / ▲, consequently this variable could not be postdicted and remains unknown.

[a] Actually there are only three possible worlds remaining because s_1^b and s_1^d are identical. However, for the purpose of illustration we do not consider this mathematical detail now.

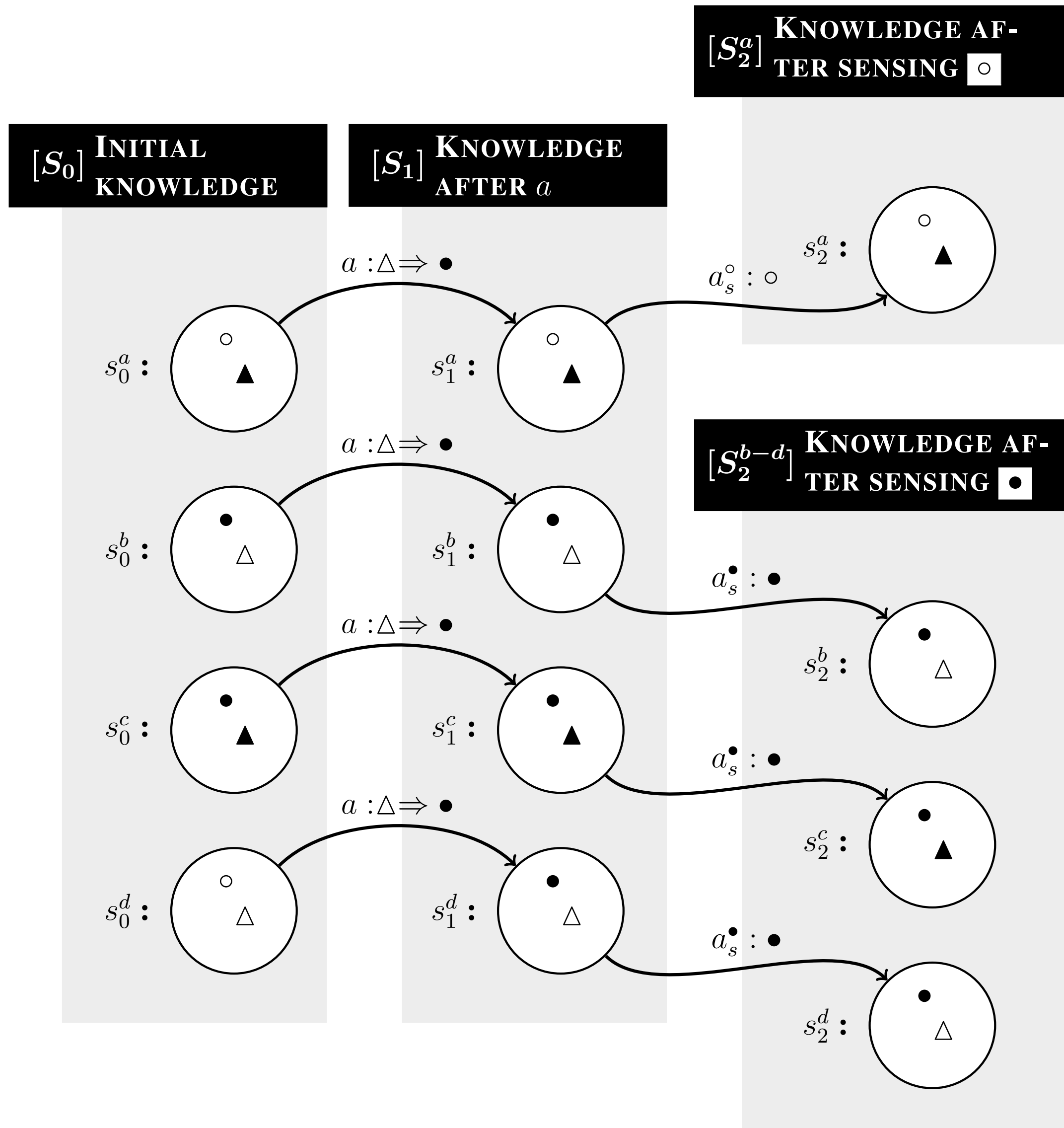


Figure 1.5.: Possible worlds model for epistemic state transitions

the action language $\mathcal{A}_k$ by Son and Baral (2001). Instead of using exponentially many possible worlds, the 0-approximation considers only one world which is the intersection of all possible worlds. It uses a 3-valued knowledge model, i.e. variables are known to be true, known to be false, or unknown. Knowledge is produced only by sensing and deductive causal reasoning, as illustrated in Example 1.2.

Example 1.2 0-Approximation Model of Knowledge
Figure 1.6 shows the transition tree for the 0-approximation. Gray coloring of a fluent symbol means that its value is *unknown*. Initially $[s_0]$, the world state is completely unknown: $[s_0] = \{\bullet, \blacktriangle\}$. After applying a, both fluents remain unknown: $[s_1] = \{\bullet, \blacktriangle\}$. Sensing creates two successor states: in s_2^a holds $\circ$ and in s_2^b holds $\bullet$. In both cases, knowledge about $\circ$ / $\bullet$ is correctly generated, but knowledge about $\triangle$ / $\blacktriangle$ is not postdicted.

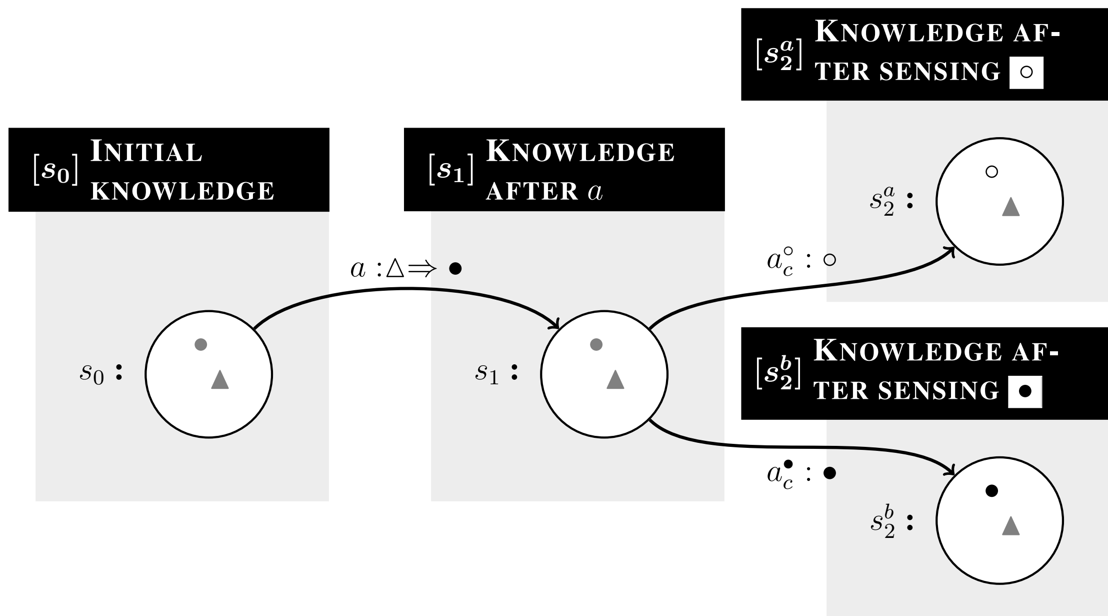


Figure 1.6.: 0-approximation model for epistemic state transitions

With this approach, modeling the knowledge state of an agent requires only $|\mathcal{F}|$ state variables and this lowers the complexity of the plan existence problem to NP-completeness (Baral et al., 2000). However, the inference capabilities of the approach are incomplete, i.e. postdiction is not supported.
Some approximations are enhanced with additional language elements like so-called Static Causal Laws (SCLs) (e.g. (Tu et al., 2007)), which can be exploited to realize postdiction in an ad-hoc manner. However, there are two major problems with this approach: first, this method is not *guaranteed to be epistemically accurate*: SCL are used to

model knowledge-level effects of actions *manually*. This implies, that a domain designer could state SCLs that have "wrong" epistemic effects. For example, an epistemically inaccurate SCL could cause to "know" that a robot is in a certain room without the robot actually being in the room.
Second, the method is not *elaboration tolerant*. The concept of elaboration tolerance was introduced by McCarthy (1998) and refers to the scalability of a theory in terms of expressiveness. In the case of $\mathcal{A}_k^c$, this means that even if the SCL is modeled in an epistemically accurate manner, accuracy can dissolve if one extends a domain, e.g. by adding more doors and rooms to the simple introductory example. Consequences of elaboration intolerance are illustrated in Example 2.1.

1.2. Contributions of this Thesis

Section (1.1) illustrated that there is a dilemma in choosing either an efficient approximate action theory with limited inference capabilities or choosing an action theory with full inference capabilities, at the cost that knowledge representation and reasoning is computationally more complex. This thesis presents the h-approximation theory that solves this dilemma. To give an overview of this contribution, we distinguish three components **C1. – C3.**

C1. We present $\mathcal{HPX}$– an approximation of the possible worlds semantics of knowledge with native support for postdiction, while the number of state-variables is linear and the planning problem is in NP. To show that $\mathcal{HPX}$ is sound wrt. traditional $\mathcal{PWS}$-based approaches we also present $\mathcal{A}_k^{TQS}$– a *temporal query semantics* for the action language $\mathcal{A}_k$ (Son and Baral, 2001) based on $\mathcal{PWS}$.

C2. We implement $\mathcal{HPX}$ in terms of Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988). The implementation as ASP is provably sound with the basic $\mathcal{HPX}$ semantics and lays the ground for the practical application.

C3. We extend the original implementation such that it is capable of performing online action planning, and we integrate the implementation in an Cognitive Robotic control framework. As a proof-of-concept and evaluation we apply the system in a Smart Home.

C1. $\mathcal{HPX}$ – an Approximate Epistemic Action Theory with a Temporal Knowledge Dimension

The *h-approximation* ($\mathcal{HPX}$) is a *history* based approximation of the $\mathcal{PWS}$ with native and elaboration tolerant support for *postdiction*. The combinatorial explosion of state variables is avoided by an alternative state representation which is not based on an

exponential number of possible worlds, but instead on a single-state world *history*. It can be understood as an extension to the 0-approximation by Son and Baral (2001), when not only the present approximated state is considered but also refinements of previous states.

Temporal Knowledge Dimension

In $\mathcal{HPX}$, the notion of *history* is used in the epistemic sense of maintaining and refining knowledge about the past by postdiction and commonsense law of inertia. That is, $\mathcal{HPX}$ considers single approximate states instead of an exponential number of possible worlds. In addition, it refines the history of these approximate states after each state transition. For instance, consider that in a world state s_0 an agent is in front of a door (denoted $\circ$) and moves forward. Later (in s_1) it acquires knowledge that it is behind the door (denoted $\bullet$); then it can postdict that the door must have been open (denoted $\triangle$) in s_0. After applying this postdiction inference it can further refine its knowledge and use the inertia assumption to infer that the door is still open in s_1. For illustration consider Example 1.3.

Linear Number of State Variables – Plan Existence in NP

Given that $|\mathcal{F}|$ is the number of fluents and t is the number of state transitions (or steps), only $|\mathcal{F}| \cdot (t+1)$ state variables are required to model an agent's "historical" knowledge state. In Section 3.3 we show that for this reason solving the *plan-existence problem* remains in NP while postdiction is still possible.

Native and Elaboration Tolerant Postdiction

According to McCarthy (1998), *"[a] formalism is elaboration tolerant to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena"*. For instance, consider a navigation problem where robots can move through doors into rooms, pick up things, etc. An epistemic side-effect in the navigation scenario is the postdictive inference of knowing that a door must be open if a robot successfully passed it. If more doors are added to the problem specification then it should not be necessary to model this side-effect for each new door (see Example 2.1 for a detailed illustration).

Concurrent Sensing and Action

Another advantage of considering the temporal dimension of knowledge is that one can elegantly model and reason about the concurrent execution of sensing and physical actions. For instance, consider the following extended version of the well-known Yale Shooting Problem (Hanks and McDermott, 1987): if an agent shoots at a turkey then it can sense whether the gun was loaded by hearing the explosion's noise. At the same

time, the bullet may kill the turkey: if the gun was loaded then it can conclude that the turkey must be dead and that the gun is unloaded after the shooting.[5] This version of the problem requires to model the shooting action with concurrent sensing (gun loaded if explosion heard) and physical effects (turkey dead, gun unloaded).
$\mathcal{HPX}$ is capable of correctly inferring that the turkey is dead if the explosion is heard because it models sensing as acquisition of knowledge about how the world is *before the action affects the world physically*. Modeling this scenario such that a conclusion about the turkey's death follows is not possible with traditional approaches because here sensing is modeled as acquisition of knowledge about how the world is *after the action affects the world*. (See Example 7.1 for details.)

$\mathcal{A}_k^{TQS}$ – A Temporal Query Semantics for $\mathcal{A}_k$

To provide a semantic grounding for $\mathcal{HPX}$ we develop a semantics which takes the role of a benchmark in terms of reasoning capabilities and expressiveness. We consider it to be epistemically complete (under some restrictions) while at the same times it allows one to make propositions about the past. The semantics is a temporal extension of the action language $\mathcal{A}_k$, called the *temporal query semantics* – $\mathcal{A}_k{}^{TQS}$.

C2. Implementation as Answer Set Programming

In order to make $\mathcal{HPX}$ accessible and applicable in real-world applications we formulate the theory in terms of Answer Set Programming (Chapter 4). A planning problem specification is formulated in a PDDL-like syntax is compiled into a Logic Program via certain translation rules. The Logic Program is processed by an off-the-shelf ASP solver which generates Stable Models (Gelfond and Lifschitz, 1988). These Stable Models can be interpreted as conditional plans for applications in Cognitive Robotics. The ASP implementation gives $\mathcal{HPX}$ an alternative model-theoretic semantics which is provably sound wrt. the operational semantics.

C3. Integration in a Cognitive Robotic Control Framework

In order to apply $\mathcal{HPX}$ in practice we implement some extensions to the original formalism. We evaluate the implementation by presenting case studies in a Smart Home.

Extensions for Online Planning and Abductive Explanation

As an extension to the basic $\mathcal{HPX}$ implementation we implement several features like execution monitoring, abductive explanatory reasoning and basic performance

[5] Under the assumption that the agent's aiming is correct.

Example 1.3 h-approximation model of knowledge
Consider Figure 1.7 which demonstrates postdictive inference with the h-approximation approach. The initial state $[S_0]$ is completely unknown: $\{\bullet, \blacktriangle\}$. Then an action a is applied which has the conditional effect that • is set if △ holds (denoted $a : \triangle \Rightarrow \bullet$). The successor state $[S_1]$ does not contain new knowledge because the condition of a is unknown (▲). $[S_1]$ consists of two sub-states which represent the present and the past world state: $s_{1:0}$ refers to the world state that the agent knows he was in before executing a and $s_{1:1}$ reflects what the agent knows about the world state after the driving. In other words, $s_{1:0}$ represents the agent's knowledge at state $[S_1]$ about how the world was at state $[S_0]$.
After execution of a a sensing action a_s is executed to acquire knowledge. We anticipate two possible outcomes of the sensing action: $[S_2^a]$ represents that state where sensing reveals that ○ holds and $[S_2^b]$ represents the state where • is the sensing result.
Sensing triggers a *refinement* of knowledge about the past: for example, in state $[S_2^a]$ the robot learns through sensing that ○ held if a_s° was executed ($s_{2:1}^a$). With this knowledge, the agent postdicts that ▲ must have held before applying a ($s_{2:0}^a$) because otherwise, due to its conditional effect, a would have caused that • holds.
The agent is also aware that no other state transition has happened that could have changed △ to ▲, i.e. △ is *inertial*. This means that △ persists to hold in $s_{2:1}^a$ and $s_{2:2}^a$ as well. Similarly, • persists to hold in all three states $s_{2:0}^a$ $s_{2:1}^a$ $s_{2:2}^a$.
In our metaphor of the robot passing through the door, this case of postdiction reflects that the robot infers closed-ness of the door (denoted ▲) because it learned by sensing that it is not behind the door (denoted ○) after the driving.
In state $[S_2^b]$ sensing reveals that • holds. However, the postdictive inference about △ / ▲ is not possible in this case, because it is not known whether or not • did already hold in $[S_0]$, respectively whether or not $\bullet \in s_{2:0}^b$. In terms of the robot-scenario, this case reflects that the robot is unable to postdict the open-state of the door because it is possible that it was already behind the door before the blindfold execution of the "move"-action.

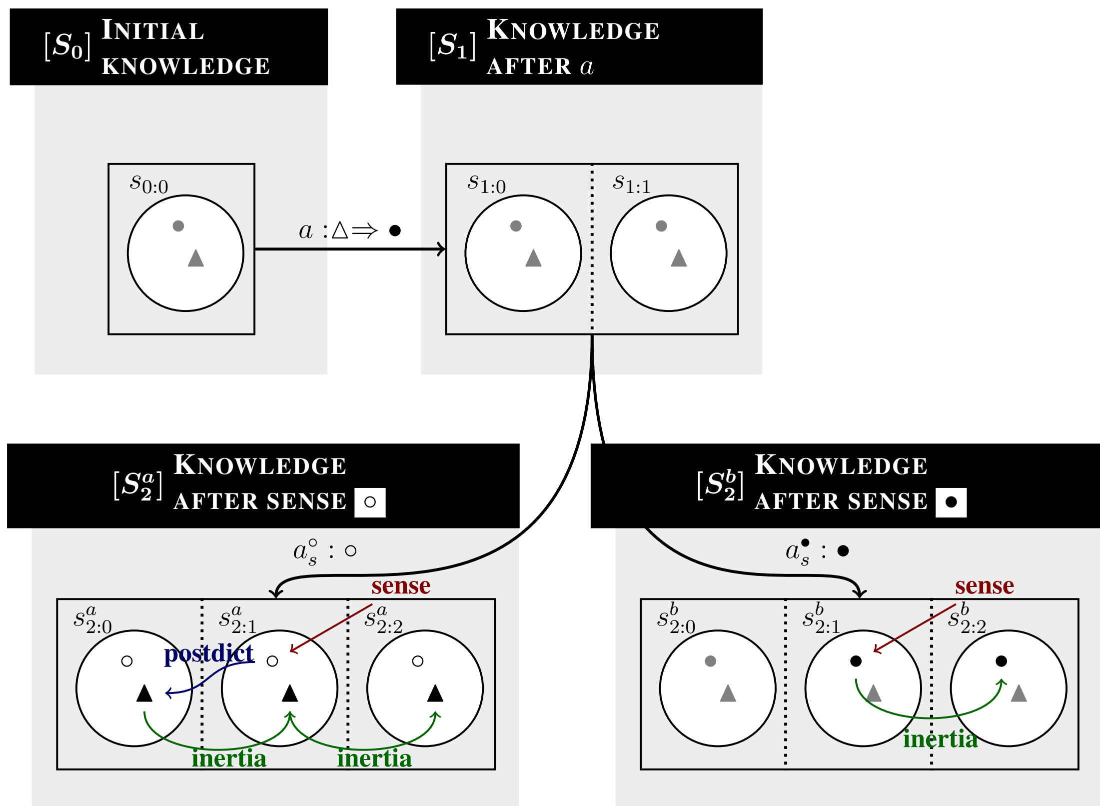


Figure 1.7.: h-approximation model for temporal epistemic state transitions

optimization (Chapter 5). These extensions are developed in response to the demands of practical applications for epistemic reasoning, such as Smart Homes, Cognitive Robotics or Narrative Interpretation tasks. The extensions are integrated as a prototypical online planning system.

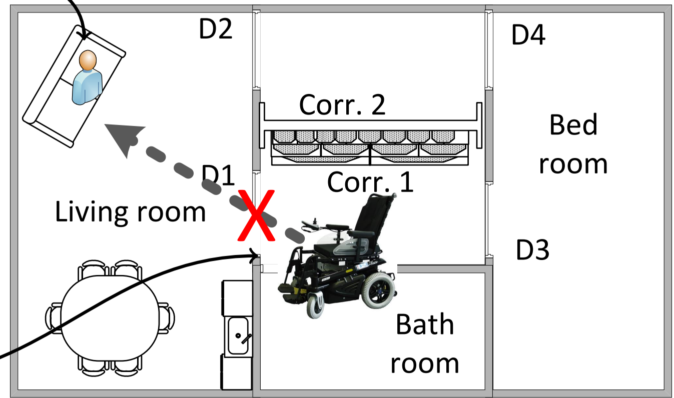
Application in a Smart Home: Postdictive Reasoning for a Wheelchair Robot

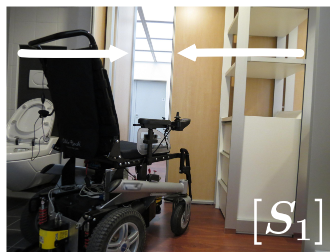
Chapter 6 contains a proof-of-concept of the work and motivates the approach with a real robotic application. We demonstrate how the $\mathcal{HPX}$ planning framework is applied in the Bremen Ambient Assisted Living Lab (BAALL) (Krieg-Brückner et al., 2010). The BAALL features many different actuators and sensors such as automatic doors, illumination control, or video cameras. BAALL's most noteworthy feature is an autonomous robotic wheelchair called "Rolland" (Mandel et al., 2005). Rolland can drive autonomously and utilizes a waypoint-based navigation module, along with obstacle-avoidance facilities (Röfer et al., 2009).
In the context of such environments, the postdiction capabilities of $\mathcal{HPX}$ are used for *abnormality detection*: abnormalities are conditions under which actions fail. For instance, a typical assistance task in the BAALL is *(a)* navigate the wheelchair to a person's location *(b)* pick the person up and *(c)* bring her to her destination. In this task, an abnormality could e.g. be caused by a jammed automatic door which can not be opened remotely anymore. If one observes that the door did not open, then one can postdict that there was an abnormality and the wheelchair has to pick an alternative route (through another door) to reach its destination.
As an example, consider Figure 1.8 which illustrates the (sub-) problem of driving the wheelchair to the sofa. Under ideal conditions, a plan is to open D1, pass it and approach the sofa. However, in real robotic environments it often happens that there is an unforeseen system failure. For instance, D1 can be blocked or jammed. In this case a more robust plan is required: open D1 and verify if the action succeeded by sensing the door status [S_1]; if the door is open, drive through the door and approach the user. Else postdict that there was an abnormality concerning the opening of D1. In this case, open and pass D3; drive through the bedroom [S_2]; pass D4 and D2; and approach the sofa [S_3].

1.3. Thesis Outline

This introduction briefly illustrates the complexity problem of traditional $\mathcal{PWS}$-based epistemic action theory. It sketches how $\mathcal{HPX}$ solves this problem, and depicts how postdiction is used in practice, e.g. for abnormality detection.
Chapter 2 communicates the basics of the related research fields *Reasoning about*

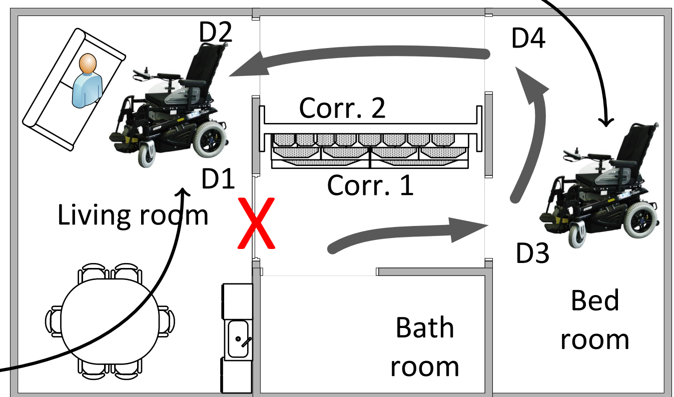
$[S_0]$: Wheelchair is called using remote control or other input device.





$[S_1]$: Door is jammed.

$[S_2]$: Wheelchair takes alternative route.



$[S_3]$: Destination reached.

Figure 1.8.: The autonomous wheelchair *Rolland* operating in the Smart Home *BAALL*

Action and Change, *Answer Set Programming*, and *Epistemic Logic*. It also places the h-approximation within current research in the field by identifying strengths and weaknesses of state-of-the-art approaches.
Chapter 3 is the core chapter of this thesis. It formalizes $\mathcal{HPX}$ in terms of a transition function semantics and describes how the temporal knowledge dimension and the postdiction mechanisms are implemented. In order to define a notion of soundness for $\mathcal{HPX}$, the chapter also provides an extended *temporal query semantics* for the action language $\mathcal{A}_k$ (Son and Baral, 2001), which allows for temporal reasoning. We prove that $\mathcal{HPX}$ is sound wrt. this extended semantics.
Chapter 4 provides the implementation of $\mathcal{HPX}$ in terms of Answer Set Programming and describes how the implementation is formally related to the operational semantics.
Chapter 5 describes the $\mathcal{HPX}$ planning framework and its implementation. The framework contains certain extensions and optimizations which allow for online planning, interleaved with abductive explanation and automated plan repair.
Chapter 6 contains an extensive case study in the Smart Home BAALL (Krieg-Brückner et al., 2012). This serves as the practical evaluation and proof-of-concept of the $\mathcal{HPX}$ planning framework.
Chapter 7 concludes this thesis. It discusses strengths and limitations of $\mathcal{HPX}$ and provides an outlook towards future work.
Appendix A contains proofs pertaining to the soundness of the ASP formalization of $\mathcal{HPX}$ wrt. its operational semantics.
Appendix B contains proofs for the computational properties of $\mathcal{HPX}$, in particular the computational complexity.
Appendix C contains soundness results of the operational $\mathcal{HPX}$-theory wrt. the extended $\mathcal{A}_k$ semantics.
Appendix D contains examples and source code of the $\mathcal{HPX}$ implementation.

2 Background and Related Work

This chapter provides preliminaries, which are required to follow the core chapters of the thesis and also encompasses related work. Section 2.1 concerns the field of *Reasoning about Action and Change* (RAC): it provides an overview over operational and model-theoretic approaches and contains definitions of the problems which $\mathcal{HPX}$ can solve: the projection problem and the planning problem. Further, we describe the inherent non-monotonicity of action theory.
Section 2.2 describes Answer Set Programming (ASP) and the Stable Model Semantics of Logic Programming.[1] The section is preliminary to Chapters 4 and 5, which describe the ASP formalization of $\mathcal{HPX}$. The section also motivates the decision to use Answer Set Programming for the implementation of $\mathcal{HPX}$.
Section 2.3 is a brief introduction to Modal Logic and the Possible Worlds Semantics of knowledge. This is preliminary to Section 2.4, which focuses on other epistemic action theories and places $\mathcal{HPX}$ within the state of the art. The latter section highlights and summarizes features of the individual approaches and compares related work accordingly.

2.1. Reasoning about Action, Change and Knowledge

The research field of *Reasoning about Action and Change* (RAC) deals with the problem of determining and formalizing how actions change the world. Research in this field can be traced back to early work by McCarthy (1959) who envisioned “Programs with

[1]Examples and notation are partly taken from the article “Answer Set Programming: A Primer” by Eiter et al. (2009) and the textbook “Answer Set Programming in Practice” by Gebser et al. (2012b). The latter describes the *Potassco ASP toolkit* which we use to implement the ASP formalization of $\mathcal{HPX}$. To describe the semantics of incremental ASP solving and online ASP solving we cite many definitions from (Gebser et al., 2008, 2011a).

Common Sense". McCarthy was primarily concerned with cases where an agent has *complete knowledge* about its domain of discourse. However, since having complete knowledge about a domain is a very strong assumption, researchers also investigated the epistemic case of *incomplete knowledge* and tried to formalize *sensing actions*. The first logical formalization which considers incomplete knowledge is due to Moore (1985) who implemented the concept of *possible worlds* from Modal Logic to action theory.
One way to formalize action and change is to use a first order logical theory, possibly with second order extensions. Examples are the Situation Calculus (McCarthy, 1963), the Fluent Calculus (Hölldobler and Schneeberger, 1990; Thielscher, 1998), the Event Calculus (Kowalski and Sergot, 1986) and Temporal Action Logic (Doherty, 1994).
Another possibility to formalize action and change is the syntactic definition of a high-level action language and to ground the language in a set-theoretic operational semantics. This approach is commonly used e.g. in action planning. The planning language PDDL (McDermott et al., 1998) is based on STRIPS (Fikes and Nilsson, 1972) and the Action Description Language (ADL) (Pednault, 1994) which are both formalized in an operational semantics.
In both cases, a domain of discourse $\mathcal{D}$ contains descriptions of actions and information about the initial world state.[2] Properties of a world, like the battery state of a robot or the open state of a door, are represented by variables called *fluents*.[3] A *fluent literal* (for brevity often simply called "literal") is a pair of a fluent and its value.
In brief, one can understand reasoning about action, change and knowledge as the problem of formalizing how fluents change over time. The main reasoning tasks in action theory are *(a) projection*, *(b) planning* and *(c) abductive explanation.*

2.1.1. Operational Semantics

An operational action-theoretic semantics typically considers a *snapshot* of the world and seeks to formalize an action's effect on this snapshot. Such snapshots are usually referred to as *states* and the occurrence of an action is understood as a *state transition*. For instance, if in a state s_0 there is a robot in a room A and the robot executes a *move*-action to an adjacent room B, then there is a transition from s_0 to a successor state s_1 such that the robot is in room B after the transition. However, the transition can depend on certain conditions, e.g. that the robot's battery is full and that there is a door between the two rooms.
A state is a set of fluent literals. For instance, a state representing that a robot's battery is full and that the robot is in room A and that the door to the room is closed may be

[2]For some action theories a domain also involves so-called State Constraints (see e.g. (Thiebaux et al., 2003)), Static Causal Laws (see e.g. (Tu et al., 2007)) or other extensions, but discussing this is not within the scope of this thesis.

[3]The variable itself is also referred to as a *feature* and called *fluent* if its change over time is considered (Sandewall, 1994). For simplicity we will use only use the term fluent in this thesis.

represented as: $s = \{\texttt{battery_full}, \texttt{in_roomA}, \neg\texttt{is_open}\}$. For non-epistemic action formalisms a state must be completely determined, i.e. the value of every fluent must be specified. This is not the case when incomplete knowledge is considered.

State Transitions

Transitions between states are modeled by a transition function which map an action and a state to a state.[4] Let $\mathcal{D}$ be a domain with a set of action symbols $\mathcal{A}$, the set of fluents $\mathcal{F}$ and the set of allowed fluent values $\mathcal{V}$. Let $\mathcal{S} \subseteq \mathcal{F} \times \mathcal{V}$ be the subset of all consistent fluent-value pairings, i.e. the set of possible *states*. Then a transition function ψ of $\mathcal{D}$ has the signature $\psi : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{S}$. If considering complete knowledge then S must be *complete*, i.e. all fluents must be assigned a value.
This simplification is not made for the case of incomplete knowledge. If considering incomplete knowledge then a transition function has to account for *sensing actions*. The occurrence of sensing actions generates *contingencies*, i.e. all possible outcomes of the sensing have to be accounted for separately.[5] For instance, consider a sensing action a_s which reveals whether or not a fluent f holds. Then projecting the sensing result of a_s on future states requires to consider one possible successor state for each possible sensing outcome f and $\neg f$. This behavior is called *branching*, i.e. each branch represents one possible sensing outcome. A transition function which considers sensing actions and branching has the signature $\psi : \mathcal{A} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$, where states in $\mathcal{S}$ may now be incomplete. For example, let `sense_open` denote an action which determines the open-state of a door (denoted `is_open`) then for a state $s = \emptyset$ we have $\psi(\texttt{sense_open}, s) = \{\{\texttt{is_open}\}, \{\neg\texttt{is_open}\}\}$.

Plans: Combinations of State Transitions

Plans (denoted p) are well-formed formulae defined in the theory's input language and denote combinations of action occurrences. To model the execution of plans one typically defines an *extended* transition function which maps a plan and a state to a state or a set of states. In the case of complete knowledge the extended transition function has a signature $\widehat{\psi} : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{S}$, where we use $\mathcal{P}$ to denote the set of well-formed plans according to a syntax specification and $\mathcal{S}$ is the set of complete consistent states.
A simple form of a plan which is often used in the case of complete knowledge is a *sequence* of actions, commonly represented in the syntactic form $[a_1; \ldots; a_n]$ with $a_1, \ldots, a_n \in \mathcal{A}$, were $\mathcal{A}$ is the set of domain actions.

[4] Some calculi like ADL (Pednault, 1994) also consider actions to be themselves functions that map states to states.

[5] Planning with incomplete knowledge is also known as *contingent planning* (see e.g. (Hoffmann and Brafman, 2005)).

For planning with incomplete knowledge one usually considers *conditional plans* which involve if-then-else constructs. For example, a plan

$$p_c = [a_s;\ \texttt{if}\ \varphi\ \texttt{then}\ a_1\ \texttt{else}\ a_2]$$

denotes that first a_s is executed and subsequently a_1 or a_2 are executed, depending on whether or not a propositional formula φ is true (see also Definition 3.5). An extended transition function which considers sensing actions has the signature $\psi : \mathcal{A} \times \mathcal{S} \to 2^{\mathcal{S}}$, where states in $\mathcal{S}$ may be incomplete.

The Projection Problem

Projection is the *deductive* reasoning task of determining possible states of the world after a plan is applied on an initial state.
In the case of incomplete knowledge one can either ask whether a world property *possibly* holds after a plan is executed or whether a world property *necessarily* holds. The first case refers to *weak projection* and the second case refers to *strong projection* (see e.g. (Cimatti et al., 2003)). Weak and strong projection are formalized in Definition 2.1.

Definition 2.1 (The projection problem for operational action theories) *Let $\mathcal{D}$ be a domain description, such that $\widehat{\psi} : \mathcal{P} \times \mathcal{S} \to 2^{\mathcal{S}}$ is a transition function of the domain and s_0 is an initial state. Let p be a plan which contains actions $\{a_1, \ldots, a_n\} \subseteq \mathcal{A}$ and let $\mathcal{G}$ be a set of fluent literals.*

- *The weak projection problem is to decide whether (2.1) holds.*

$$\exists s \in \widehat{\psi}(p, s_0) : \mathcal{G} \subseteq s \tag{2.1}$$

- *The strong projection problem is to decide whether (2.2) holds.*

$$\forall s \in \widehat{\psi}(p, s_0) : \mathcal{G} \subseteq s \tag{2.2}$$

The Planning Problem

Planning is a decision-theoretic method based on the motivation to deliberate agents in the task of achieving a specified goal $\mathcal{G}$ starting in an initial wold state s_0. One is interested in finding a plan p, such that the execution of the plan causes a transition from state s_0 to a final state s' such that $\mathcal{G} \subseteq s'$.
For *weak planning* one is interested in whether a goal is entailed in at least one possible leaf state and for *strong planning* a goal must hold in all possible leaf states. This is described in Definition 2.2.

Definition 2.2 (The planning problem for operational action theories) *Let $\mathcal{D}$ be a domain description, such that $\widehat{\psi} : \mathcal{P} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$ is a transition function of the domain and s_0 is an initial state. Let p be a plan which contains actions $\{a_1, \ldots, a_n\} \subseteq \mathcal{A}$ and let $\mathcal{G}$ be a set of fluent literals.*

- *The* weak *planning problem is that deciding whether (2.3) holds.*

$$\exists p : \exists s \in \widehat{\psi}(p, s_0) : \mathcal{G} \subseteq s \tag{2.3}$$

- *The* strong *planning problem is that deciding whether (2.4) holds.*

$$\exists p : \forall s \in \widehat{\psi}(p, s_0) : \mathcal{G} \subseteq s \tag{2.4}$$

Abductive explanation

Abductive explanation is a diagnostic method which seeks to find a cause for why the world is as it is. Technically, abductive explanation is based on the same reasoning mechanism as planning with complete knowledge: given an initial state s_0 and a set of world properties $\mathcal{G}$ one is interested in a course of action that can explain why the world changed from s_0 to $\mathcal{G}$. The difference between planning with complete knowledge and abductive explanation lies solely in the application one is interested in. For abductive explanation a set of world properties $\mathcal{G}$ is known to hold at present or in the past, while for planning, $\mathcal{G}$ is a goal which one seeks to achieve in the future. For instance, abductive explanation may be used in forensic reasoning to find out how an object was stolen, while in planning one would be interested in a way to steal the object.
Despite the technical equivalence of action planning and abductive explanation, there are use cases where both reasoning tasks are performed in an interleaved manner and where it is important to distinguish both tasks. In Section 6.2 we present a scenario which underpins this observation.

2.1.2. Model-theoretic Semantics

As an alternative to operational semantics one can also use a model-theoretic semantics to formalize the problems of projection, planning and abductive explanation. Respective action theories are usually specified in First Order Logic (FOL), possibly with some second-order extensions. Examples are the Situation Calculus (McCarthy, 1963), the Event Calculus (Kowalski and Sergot, 1986), and Temporal Action Logic (Doherty, 1994). Here, a domain $\mathcal{D}$ is considered as a first-order theory which represent the initial world state and how actions affect the world. To denote that a fluent f is true after the t-th state transition one typically uses a predicate $holds(f, t)$, or in the epistemic case $knows(f, t)$. Branching is realized by using predicates with an additional parameter

which represents a label for the branch. For example, $holds(f, t, b)$ denotes that f is true after t state transitions in a branch b.
The intuition behind the problems of projection, planning and abductive explanation is the same as for operational semantics.

The projection problem for model-theoretic semantics

For the projection problem, a plan Γ_p is given as a conjunction of logical facts, and one is interested in whether a formula φ_g is logically entailed the conjunction of the domain theory and Γ_p. In the case of incomplete knowledge, φ_g usually contains predicates which involve a parameter b to denote the label of a branch.

Definition 2.3 (The projection problem for model-theoretic action theories) *Let $\mathcal{D}$ be a first-order logical domain theory and Γ_p be a set of formulae which denote the occurrence of actions and let $\varphi_g(b)$ denote a formula which involves predicates with a parameter b.*

- *The weak projection problem is to decide whether (2.5) holds.*

$$\exists b : \mathcal{D} \wedge \Gamma_p \models \varphi_g(b) \tag{2.5}$$

- *The strong projection problem is to decide whether (2.6) holds.*

$$\forall b : \mathcal{D} \wedge \Gamma_p \models \varphi_g(b) \tag{2.6}$$

The planning problem for model-theoretic semantics

For the planning problem, a formula φ_g is given and one seeks to find a plan Γ_p such that φ_g is entailed in the logical theory $\Gamma_p \wedge \mathcal{D}$.

Definition 2.4 (The planning problem for model-theoretic action theories) *Let $\mathcal{D}$ be a first-order logical domain theory and $\varphi_g(b)$ be a formula denoting a goal state which involves predicates with a parameter b.*

- *The* weak *planning problem is to decide whether (2.7) holds.*

$$\exists \Gamma_p : \exists b : (\mathcal{D} \wedge \Gamma_p \models \varphi_g(b)) \tag{2.7}$$

- *The* strong *planning problem is to decide whether (2.8) holds.*

$$\exists \Gamma_p : \forall b : (\mathcal{D} \wedge \Gamma_p \models \varphi_g(b)) \tag{2.8}$$

2.1.3. Non-monotonicity and Circumscription

Action theories are usually non-monotonic. In general, a theory is monotonic if given a knowledge base KB and two formulae α and β the following holds:

$$\text{if } KB \models \alpha \text{ then } KB \cup \beta \models \alpha. \tag{2.9}$$

If (2.9) does not hold, then a theory is *non-monotonic*.

Non-monotonicity in Action Theory

Action theories are inherently non-monotonic if the *inertia assumption* is made. The assumption presumes that a world property persists if nothing happened that changed this property. This implies that one considers a *closed world* where the agent is aware of all actions that happen and where everything that is not known to be true or false is assumed to be false. Consider the following example from the blocksworld domain (Winograd, 1971): an action theory $\mathcal{D}^B$ represents that a block is on the table in a situation S_0. The block will persist on the table in situation S_1 if it is not picked up in S_0.

$$\begin{aligned}\mathcal{D}^B =& OnTable(Block, S_0) \\ &\wedge(OnTable(Block, S_1) \Leftrightarrow OnTable(Block, S_0) \wedge \neg PickUp(Block, S_0))\end{aligned} \tag{2.10}$$

Under the closed world assumption $\neg$ means "can not be shown to be true". Consequently:

$$\mathcal{D}^B \models OnTable(Block, S_1)$$

If we conjoin $\mathcal{D}^B$ with $PickUp(Block, S_0)$ then we see that the theory is non-monotonic:

$$\mathcal{D}^B \wedge PickUp(Block, S_0) \not\models OnTable(Block, S_1)$$

Circumscription and the Frame Problem

One way to implement non-monotonicy is *Circumscription* (McCarthy, 1980). Circumscription reflects the closed-world assumption, i.e. everything that is not known to be true is considered to be false. The most common applications of circumscription in action theory are the implementation of the law of inertia and reasoning about abnormalities. Intuitively, circumscription minimizes the extension of a predicate in a theory. (Lifschitz, 1994) formulates circumscription with the following second-order definition:

Definition 2.5 (Circumscription) *Let $\Phi(P)$ be a formula containing a predicate constant P. Let Q be a predicate variable with the same arity as P, and $\Phi(Q)$ denote the formula where all occurrences of P in Φ are replaced by Q. Then the circumscription of*

Φ that minimizes predicate P, denoted as $CIRC[\Phi; P]$, is a sentence schema represented by the following second-order formula:

$$
\begin{aligned}
&\Phi(P) \wedge \neg\exists Q : [\Phi(Q) \wedge Q < P] \\
&\textit{where} \\
&Q = P \textit{ means } \forall \vec{x} : [Q(\vec{x}) \Leftrightarrow P(\vec{x})] \\
&Q \leq P \textit{ means } \forall \vec{x} : [Q(\vec{x}) \Rightarrow P(\vec{x})] \\
&Q < P \textit{ means } [Q \leq P] \wedge \neg[Q = P]
\end{aligned}
\tag{2.11}
$$

Equation (2.11) can be understood as a condition under which a first-order theory is circumscribed.
Circumscribed theories contain a number of axioms which intuitively provide a "frame" that limits the extent of each fluent wrt. each action. In the Situation Calculus (McCarthy and Hayes, 1969) such axioms are called *Frame Axioms*. The problem with this approach is that the number of such axioms is usually huge: if there are $|F|$ fluents and $|A|$ actions, then a properly circumscribed theory contains $O(|A| \cdot |F|)$ frame axioms (see e.g. (McCarthy and Hayes, 1969)). This problem is commonly known as the *Representational Frame Problem*. Though there are partial solutions to this problem, such as the successor state axioms described in (Reiter, 1991), circumscription always has the disadvantage that a domain description can not be arbitrary. It must be "compatible" with the circumscription formula (2.11).
An alternative to circumscription is using formalisms which are based on the so-called Negation as Failure (NaF) principle. One formalism which follows this principle is Answer Set Programming.

2.2. Answer Set Programming

Answer Set Programming (ASP) is a form of Logic Programming which uses Negation as Failure (NaF) to implement non-monotonic reasoning.
Other approaches to Logic Programming, in particular Prolog, use an algorithm called *SLDNF-resolution* (Kowalski, 1974). The disadvantage of these approaches is that SLDNF-resolution is not fully declarative: in particular conjunction is not commutative and the order in which rules are written has an influence of the evaluation of a Logic Program. For instance, if rules are not provided in an appropriate order then SLDNF might not terminate.
Answer Set Programming (ASP) is an alternative approach to Logic Programming which does not have these limitations. ASP is based on the *Stable Model* (SM) semantics (Gelfond and Lifschitz, 1988) which makes ASP fully declarative. ASP solvers take Logic Programs (LPs) as input and employ the Stable Model Semantics to find solutions to Logic Programs which are called *Answer Sets*. To explain how Answer Sets are

computed for so-called *normal* Logic Programs we first explain the simpler case of *positive* Logic Programs.

2.2.1. Positive Logic Programs

Positive Logic Programs are LPs that do not contain negations. A positive LP P is defined by a set of rules of the form

$$h \leftarrow b_1, \ldots, b_n. \tag{2.12}$$

with $0 \leq n$ and where h and $b_1, \ldots, b_n$ are symbols called *atoms*. h is called the head and $b_1, \ldots, b_n$ is called the body of a rule r, denoted $head(r)$ and $body(r)$ respectively. If the body of a rule is empty ($n = 0$) then the arrow symbol $\leftarrow$ is typically omitted and the rule is called a *fact*.

Solving Positive Logic Programs

ASP pursues a bottom-up approach to solve Logic Programs in that it a-priori considers all subsets of a set of atoms to be possible solutions, called *interpretations* of the Logic Programs. For instance, consider the following Logic Program:

$$\begin{aligned} a &\leftarrow b. \\ b &\leftarrow a. \\ &c. \end{aligned} \tag{2.13}$$

The set of possible interpretations of (2.13) is given by the powerset $2^{\{a,b,c\}}$. In a "filtering" step, the ASP solver rules out all these interpretations which are not "compatible" with the constraints and facts defined in the program. For instance, an interpretation $\{a, b\}$ is not compatible because c is a fact which must also be true but which is not contained in the set. What is left after ruling out incompatible interpretations are the *models* of P, denoted by M. In the above case these are $\{c\}, \{a, b, c\}$. In Logic Programming, one is interested in these models which are *minimal* in the sense of Definition 2.6.

Definition 2.6 (Minimal Model of a Logic Program) *A model M of a Logic Program P is a Minimal Model if there is no other model M' of P such that $M' \subset M$.*

It turns out that every normal Logic Program has exactly one minimal model, as stated by Theorem 2.1.

Theorem 2.1 (Least Model of a Logic Program) *Every positive Logic Program P has a single minimal model, also called the* Least Model*, denoted by $LM(P)$.*

A proof of the theorem can be found in literature, e.g. (Eiter et al., 2009). The Least Model of a positive LP P, denoted by $LM(P)$ can be computed iteratively with a *consequence operator* T_P. Let I be an interpretation of P, then

$$T_P(I) = \{a | \exists r \in P : head(r) = a \wedge body(r) \subseteq I\} \tag{2.14}$$

Positive Logic Programs have a least fixpoint, as stated by Theorem 2.2.

Theorem 2.2 (Least fixpoint of a positive Logic Program) *Let $T_P^0 = \emptyset$ and $T_P^{i+1} = T_P(T_P^i)$ with $i \geq 0$. Then T_P has a least fixpoint $lf(T_P)$ to which T_P^i converges for $i \geq 0$, such that $lf(T_P) = LM(P)$.*

A proof can be found e.g. in (Eiter et al., 2009).

2.2.2. Grounding

So far we considered *grounded* Logic Programs. A Logic Program P is grounded if predicates do not contain variables, i.e. if predicates are atoms. In the general case however, predicates do contain variables which are usually denoted by uppercase letters. We call Logic Programs which contain variables *non-ground* Logic Programs, and write $grd(P)$ to denote a grounded version of a non-ground Logic Program P. Consider the Logic Program (2.15) as an example which implements a simple blocksworld domain. The first rule in (2.15) states that for any robot, any object and any location in the domain of discourse, if a robot is holding an object, then the object must be at the same location as the robot.

$$\begin{aligned} at(O,X) &\leftarrow holding(R,O), at(R,X), robot(R), location(X), object(O).\\ robot(pr2)&.\\ object(block_1)&.\\ object(block_2)&.\\ location(atTable)&. \end{aligned} \tag{2.15}$$

Variables are implicitly universally quantified over a set $\mathcal{C}$ of constants that are declared in the Logic Program as lower-case arguments of predicates. In the above LP we have $\mathcal{C} = \{pr2, block_1, block_2, atTable\}$:

Before solving a LP it has to be *grounded*, i.e. variables have to be eliminated. Software tools like *gringo* (Gebser et al., 2011b) employ efficient algorithms for this purpose. The

grounded version of LP (2.15) is the LP (2.16).

$$\begin{aligned}
&robot(pr2).\\
&object(block_1).\\
&object(block_2).\\
&location(atTable).\\
&at(block_1, atTable) \leftarrow holding(pr2, block_1), at(pr2, atTable),\\
&\qquad robot(pr2), location(atTable), object(block_1).\\
&at(block_2, atTable) \leftarrow holding(pr2, block_2), at(pr2, atTable),\\
&\qquad robot(pr2), location(atTable), object(block_2).
\end{aligned} \tag{2.16}$$

Logic Programs can also contain functions which may be arguments of predicates or other functions. For instance, the following rule is a reified version of the first rule in (2.15) which involves a modularity S. S denotes a *situation* in which a fact holds:

$$\begin{aligned}
holds(at(O, X), S) \leftarrow &holds(holding(R, O), S), holds(at(R, X), S),\\
&robot(R), location(X), object(O), situation(S).
\end{aligned} \tag{2.17}$$

In (2.17) the binary predicate $holds$ has as argument the binary functions at and $holding$ plus a variable S.

2.2.3. Normal Logic Programs and Negation as Failure

Normal Logic Programs are more general than positive Logic Programs because they can contain negated atoms in the bodies of their rules. A grounded normal Logic Program P is defined by a set of rules of the form

$$h \leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_{n+m}. \tag{2.18}$$

with $0 \leq n \leq n + m$.

The symbol not denotes *Negation as Failure* (NaF), also known as *default negation* or *weak negation*. In the following, an atom a or the default negation $not\ a$ of an atom is referred to as a *literal*. It is important to not confuse default negation with *classical negation* (denoted by "~") which is discussed in Section 2.2.4.

Intuitively, NaF is a form of "sceptical" reasoning which means that something is considered not to hold if the formalism fails to prove that it holds. For instance, the Logic Program

$$\begin{aligned}
&c.\\
a \leftarrow\ &b.\\
b \leftarrow\ &a.
\end{aligned} \tag{2.19}$$

has one Stable Model c. The atoms a and b are not in the Stable Model, because there is no *evidence* that they hold. In contrast, if we treat the program (2.19) as a formula in Propositional Logic, i.e.

$$c \wedge (b \Rightarrow a) \wedge (a \Rightarrow b) \tag{2.20}$$

then, because Propositional Logic does not employ Negation as Failure, we obtain two models: $c \wedge a \wedge b$ and c.
The Negation as Failure principle emerges from the Stable Model semantics (Gelfond and Lifschitz, 1988). In Section 2.2.6 we describe how this is used to realize non-monotonic reasoning in action theory.

The Stable Model Semantics

Normal Logic Programs underly the so-called *Stable Model Semantics*. The definition of a *Stable Model* is based on the Gelfond-Lifschitz reduct (Gelfond and Lifschitz, 1988) of a Logic Program P wrt. a model M. This is described in Definition 2.7.

Definition 2.7 (The Gelfond Lischitz Reduct) *Let $body^+(r)$ be the set of positive literals in a rule r of a Logic Program, and $body^-(r)$ the set of negative literals. The Gelfond-Lifschitz reduct (GL-reduct) of a Logic Program P wrt. a model M, denoted by P^M, is defined as:*

$$P^M = \{head(r) \leftarrow body^+(r) | r \in P \wedge body^-(r) \cap M = \emptyset\} \tag{2.21}$$

The reduct of a program P wrt. a model M, denoted P^M, is obtained as follows: *(a)* For all atoms $a \in M$ remove every rule which contains $not\ a$ in its body, so that the rule can not trigger its head atom to be true. *(b)* For the remaining rules remove all negative literals from their bodies. These can be assumed true as their atoms are not in M anyways.
Obviously, a reduct P^M is always a positive Logic Program as it does not contain negative literals anymore. Hence, with Theorem 2.1, a reduct must always have one unique Least Model $LM(P^M)$. This is said to be a *Stable Model* of P if it is equal to the original model M. The set of Stable Models of a Logic Program is denoted by $SM[P]$. Formally:

Definition 2.8 (Stable Model) *A model M of a Logic Program P is a* Stable Model *of P if it is equal to the Least Model of the Gelfond-Lifschitz reduct $LM(P^M)$:*

$$M \in SM[P] \Leftrightarrow M = LM(P^M) \tag{2.22}$$

The definition of Stable Models implies that a normal LP can have multiple Stable Models, also called *Answer Sets*. Here, a weakly negated literal causes a reduct not to contain rules with the negated literal. Therefore these literals cause models to be "unstable". Alternative approaches such as the *Perfect Model* semantics (Przymusinski, 1987) treat weak negation differently, such that only one model (the Perfect Model) for a normal Logic Program is allowed.

2.2.4. Extensions to the Stable Model Semantics

Several extensions to the Stable Model Semantics were proposed and implemented to increase the expressiveness of Answer Set Programming and to facilitate the Logic Program design. In the following, a non-exhaustive summary about extensions implemented in the *Potassco* ASP toolkit (Gebser et al., 2012b) is provided. The described extensions are preliminary to the ASP formalization of $\mathcal{HPX}$ in Chapter 4.

Integrity Constraints

Default negation can be used to formulate so-called *integrity constraints*. These are rules where the head is empty, i.e. rules of the following form:

$$\leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_{n+m} \tag{2.23}$$

Integrity Constraints do not require a semantical extension and can be rewritten using Negation as Failure as follows:

$$h \leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_{n+m}, not\ h. \tag{2.24}$$

Intuitively, an integrity constraint of the form (2.24) can be understood as "filter" which rules out these Stable Models which do not contain atoms $\leftarrow b_1, \ldots, b_n$ and which do contain $b_{n+1}, \ldots, b_{n+m}$.

Strong Negation

Strong negation is usually represented with the "~"-symbol. While Negation as Failure (NaF) intuitively means that it can not be proven that a fact holds, strong negation means that it can be proven that a fact does not hold. Gelfond (1994) pointed out that similar to autoepistemic logic, strong negation can be understood as *knowing that something does not hold*, while NaF refers to not knowing that something holds. Consider the following example which can be found in (Gelfond and Lifschitz, 1991):

$$canPass(L) \leftarrow railroadCrossing(L), not\ trainApproaching(L). \tag{2.25}$$

$$canPass(L) \leftarrow railroadCrossing(L), {\sim}trainApproaching(L). \tag{2.26}$$

Here, (2.25) means hat if it is not known that a train is approaching, then the railroad crossing can safely be passed. In contrast, (2.26) is a stronger and "safer" assertion: It means that the railroad crossing can only be passed if it is known that no train is approaching.

Strong negation does not require an extension of the Stable Model semantics. It can be compiled away by treating a strongly negated atom $\sim a$ as a new independent atom and by adding the integrity constraint (2.27) for every atom a in the Logic Program.

$$\leftarrow a, {\sim}a. \tag{2.27}$$

Basic Arithmetic

State-of-the-art ASP grounders like *gringo* (Gebser et al., 2011b) support basic arithmetic with integer numbers. For instance, consider the following simple program:

$$\begin{aligned} int(0..10). \\ sum(A, B, A + B) \leftarrow int(A), int(B). \end{aligned} \tag{2.28}$$

The first rule is a macro to define the facts $int(0). \ \ldots int(10)$. The second rule uses addition to define the sum of A and B. Apart from addition, subtraction, multiplication and division, *gringo* also supports comparison, i.e. the operators $\{<, >, \leq, \geq, =\}$.

Choice Rules

Choice Rules are used in the so-called *generation part* of a Logic Program (see e.g. (Gebser et al., 2012b)). Intuitively, Choice Rules propose candidate sets of atoms which "generate" a Stable Model if they are compatible with the other rules and constraints in the Logic Program and if they result in a model that is *stable*. Choice Rules are constructs of the form (2.29).

$$\{h_1, \ldots, h_n\} \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, not\ b_{m+k}. \tag{2.29}$$

They can be compiled into $2m + 1$ rules as follows:

$$\begin{aligned} & h' \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, not\ b_{m+k}. \\ & h_1 \leftarrow h', not\ h'_1 \\ & \ \vdots \\ & h_n \leftarrow h', not\ h'_n \\ & h'_1 \leftarrow not\ h_1 \\ & \ \vdots \\ & h'_n \leftarrow not\ h_n \end{aligned} \tag{2.30}$$

where $h', h'_1, \ldots h'_n$ are additional auxiliary atoms.

As an example consider the program (2.31).

$$\begin{aligned} a. \\ \{b, c, d\} \leftarrow a. \end{aligned} \tag{2.31}$$

The program generates 8 Stable Models $SM(P) = \{\{a\}, \{a, b\}, \{a, c\}, \{a, d\}, \{a, b, c\}, \{a, c, d\}, \{a, b, d\}, \{a, b, c, d\}\}$, that is the choice rule generates all possible combinations of facts enlisted in its head if its body is a fact.

Cardinality Constraints

Choice rules can be augmented with so-called cardinality constraints. Consider the program (2.32).

$$\begin{aligned} a. \\ 2\{b, c, d\}3 \leftarrow a. \end{aligned} \tag{2.32}$$

This program generates only these subsets of the powerset of $\{a, b, c\}$ which contain at least 2 and at most 3 elements. That is, cardinality constraints allow one to define upper and lower bounds for the number of atoms that are produced with a choice rule. However, cardinality constraints can also be used in the body of a rule, as shown in (2.33).

$$\leftarrow 2\{b, c, d\}3. \tag{2.33}$$

The rule is an integrity constraint which removes all Stable Models which contain less than 2 and more than 3 atoms of the set $\{a, b, c\}$.
Cardinality rules are a purely syntactical extension of *gringo*'s input language. The translation relies on a special counter-predicate which is explained in detail in (Gebser et al., 2012b).

Conditions

Non-grounded programs can contain *conditions* within the head of a choice rule, indicated with the ":" symbol as in (2.34).

$$\begin{aligned} &int(0..4). \\ &\{p(T) : int(T)\}. \\ &\text{where } int(0..4). \text{ is short for } int(1), int(2), int(3), int(4). \end{aligned} \tag{2.34}$$

The ":" symbol indicates a so-called *condition statement*. It causes the grounder to rewrite the head of the choice rule as a list of all possible instantiations p. That is, the grounded version of the above program is (2.35).

$$\begin{aligned} &int(0..4). \\ &\{p(1), p(2), p(3), p(4)\}. \end{aligned} \tag{2.35}$$

Note that the occurrence of variables in a condition statement has a different effect than the occurrence of variables in the body of a choice rule. For example consider (2.36) which is a variant of (2.34).

$$\begin{aligned} &int(0..4). \\ &\{p(T)\} \leftarrow int(T). \end{aligned} \tag{2.36}$$

The grounded set of rules of this variant (and hence also its Answer Sets) differs from the grounded LP of (2.34 as shown in (2.37).

$$\begin{aligned} &int(0..4). \\ &\{p(1)\}. \\ &\{p(2)\}. \\ &\{p(3)\}. \\ &\{p(4)\}. \end{aligned} \tag{2.37}$$

Optimization Statements

Modern ASP solvers have inherent support for solving optimization problems. The keywords $\#minimize$ and $\#maximize$ can be used to select one Stable Model among the set of Stable Models in a Logic Program which contains the minimal or maximal number of a certain predicate. It can also be used in combination with arithmetics to maximize the value of a variable. An obvious application for action planning is to minimize the length or cost of a plan.
The general form of a *minimization* statement of *gringo*'s input language is:

$$\#minimize\{l_1 = w_1@p_1, \ldots, l_n = w_n@p_n\} \tag{2.38}$$

Here, l_i are literals, w_i are integer numbers denoting weight and p_i are positive integers denoting *priority* for $0 \leq i \leq n$. The semantics of this statement is as follows. Let X, Y be Stable Models of a Logic Program P.
Consider a priority value p_i. Then $\sum_{p_i}^{X} = \sum_{l_i \in X} w_i$ denotes the sum of weights of a Stable Model. Let p_{max} be the maximal priority value. Then a Stable Model X is said to be *dominated* by X' if $\sum_{p_{max}}^{X'} < \sum_{p_{max}}^{X}$. If a Logic Program contains a minimization statement, then the ASP solver returns only these Stable Models which are not dominated by another Stable Model.
Maximization is an alternative form of optimization. However, this is a merely syntactic extension and specified as in (2.39).

$$\#maximize\{l_1 = w_1@p_1, \ldots, l_n = w_n@p_n\} \tag{2.39}$$

A maximization statements of the form (2.39) is equivalent to the minimization statement (2.40).

$$\#minimize\{l_1 = -w_1@p_1, \ldots, l_n = -w_n@p_n\} \tag{2.40}$$

2.2.5. Computational Properties

The computational properties of ASP are well-known. The following Theorem considers the complexity of grounded positive Logic Programs:

Theorem 2.3 (Complexity for positive Logic Programs) *Deciding whether an atom a is contained in a Stable Model of a grounded positive Logic Program is P-complete.*

Similarly for grounded normal Logic Programs:

Theorem 2.4 (Complexity for normal Logic Programs) *Deciding whether an atom a is contained in a Stable Model of a grounded normal Logic Program is NP-complete.*

And for grounded normal Logic Programs with optimization statements:

Theorem 2.5 (Complexity for normal Logic Programs with optimization) *Deciding whether an atom a is contained in a Stable Model of a grounded normal Logic Program with optimization statements is Δ_2^P-complete.*

Details and proofs for Theorems 2.3 – 2.5 can be found in literature, e.g. (Gebser et al., 2012b).

2.2.6. ASP-based Action Theory and Negation as Failure

Answer Set Programming offers a convenient solution to the frame problem described in Section 2.1.3 because it is based on the Negation as Failure Principle. This means that circumscription or similar approaches are not required to model the inertia law. As an example consider the simple block domain $LP(\mathcal{D}^B) = (2.41)$. [6]

$$\begin{aligned} onTable(block, s_0).& \\ onTable(block, s_1) &\leftarrow onTable(block, s_0), not\ noninertial(block, s_0). \\ \sim onTable(block, s_1) &\leftarrow \sim onTable(block, s_0), not\ noninertial(block, s_0). \\ noninertial(block, s_0) &\leftarrow pickUp(block, s_0). \\ \sim onTable(block, s_1) &\leftarrow pickUp(block, s_0). \end{aligned} \tag{2.41}$$

The Logic Program has one Stable Model:

$$SM[LP(\mathcal{D}^B)] = \{onTable(block, s_0), onTable(block, s_1)\}$$

To see that the reasoning is non-monotonic we add the fact $pickUp(block, s_0)$ and obtain:

$$SM[LP(\mathcal{D}^B) \cup pickUp(block, s_0)] = \{onTable(block, s_0), \sim onTable(block, s_1)\}$$

[6] Recall that opposed to First Order Logic, the convention for Logic Programming is that constants and predicate names usually start with a lower-case letter and variables start with an upper-case letter.

This approach of modeling inertia is e.g. used in the ASP formalization of the action language $\mathcal{A}$ (Gelfond and Lifschitz, 1993). To understand the advantage of this approach before circumscription (see Section 2.1.3) observe that one atom $noninertial(f, s)$ is defined for each fluent f and situation s. Therefore two rules per fluent are sufficient to model under which condition a fluent or its negation is non-inertial. Hence, given that $|\mathcal{E}|$ is the number of action effects and $|\mathcal{F}|$ is the number of fluents in the domain only $O(|\mathcal{E}| + 2|\mathcal{F}|)$ rules are required to model inertia, where $O(|\mathcal{A}| \cdot |\mathcal{F}|)$ frame axioms are required if using circumscription ($|\mathcal{A}|$ being the number of actions). It was shown that despite the lower representational complexity, the Stable Model Semantics and Circumscription are equivalent in terms of computational soundness and completeness for certain *canonical formulae* (see e.g. (Lee and Palla, 2009)).
Apart from the language $\mathcal{A}$ byGelfond and Lifschitz (1993) there are many other Action Languages which can be understood as extensions of $\mathcal{A}$ and which have been translated to ASP. For instance, $\mathcal{B}$ (Gelfond and Lifschitz, 1998, Section 5) extends $\mathcal{A}$ with indirect action effects and ramifications. $\mathcal{C}$ (Giunchiglia and Lifschitz, 1998) is another extension which also involves indirect effects, but has a more general model of inertia and allows for concurrency. $\mathcal{C}+$ (Giunchiglia et al., 2004) is an extension of $\mathcal{C}$ which is based on *universal causation* (Turner, 1999).

2.2.7. ASP Module Theory

Modularity of Answer Set Programming was defined by Oikarinen and Janhunen (2006). Intuitively, the module theory describes how separate Logic Programs can be conjoined such that the union of the answer sets of each individual LP equals the answer set of the union of the Logic Programs. We make use of the module theory in Chapter 5, where we describe how we employ *oclingo* for our iterative online ASP solving approach.
A logic program module is formally described by Definition 2.9

Definition 2.9 (Logic program module (Oikarinen and Janhunen, 2006)) *A triple* $\mathbb{P} = \langle P, I, O \rangle$ *is a (propositional logic program) module, if*

1. P *is a finite set of rules of the form* $h \leftarrow B^+, not\ B^-$*;*

2. I *and* O *are sets of propositional atoms such that* $I \cap O = \emptyset$*; and*

3. $head(P) \cap I = \emptyset$

We write $P(\mathbb{P})$*,* $I(\mathbb{P})$*,* $O(\mathbb{P})$ *to denote the constituents of a module* $\mathbb{P}$*.*

To define composability we first describe the *positive dependency graph* of a Logic Program in Definition 2.10.

Definition 2.10 (Positive dependency graph of a Logic Program) *Let P be a ground normal Logic Program. The vertices of the positive dependency graph of P are the atoms occurring in P. The edges of the graph are described by the set*

$$\{\langle a,b\rangle \,|r \in P, a \in head(r), b \in body(r)^{+}\}$$

where $body(r)^{+}$ denotes the positive atoms of a rule r.

Definition 2.11 describes composability of modules, i.e. the definition of their join.[7]

Definition 2.11 (Join of LP modules (Gebser et al., 2011a)) *Let $\mathbb{P}_1 = \langle P_1, I_1, O_1\rangle$ and $\mathbb{P}_2 = (\langle P_2, I_2, O_2\rangle$ be modules such that*

1. *$O_1 \cap O_2 = \emptyset$;*
2. *there is no edge in the positive dependency graph of $P_1 \cup P_2$ that shares atoms with both O_1 and O_2.*

Then the join of $\mathbb{P}_1$ and $\mathbb{P}_2$, denoted as $\mathbb{P}_1 \sqcup \mathbb{P}_2$, is defined as the module

$$\langle P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1)\,, O_1 \cup O_2\rangle$$

Answer Sets of modules are described in Definition 2.12.

Definition 2.12 (Answer Sets of Modules (Gebser et al., 2011a)) *A set of atoms X is an Answer Set of a module $\mathbb{P} = \langle P, I, O\rangle$ if X is an Answer Set of $P \cup \{a \leftarrow |a \in X \cap I\}$. We denote the set of answer sets of a module $\mathbb{P}$ as $AS(\mathbb{P})$.*

2.2.8. Iterative ASP Solving

Iterative ASP solving is used to perform *slice-wise* (Gebser et al., 2011b) grounding of the Logic Program according to a single integer iterator t. Which each iteration a new set of rules (also called a *slice*) is added to the grounded LP. Each slice is a modular element of the Logic Program.

Incremental Modularity

If using the incremental ASP solver *iclingo* (Gebser et al., 2011b), incremental problem solving is realized by splitting a LP into three parts: $\#base$, $\#cumulative$ and $\#volatile$. The $\#base$ part is a part of the Logic Program which does not contain the iterator t. The $\#cumulative$ part includes rules with the parameter t and the set of rules which is instantiated for each t accumulates with the previously grounded LP. The $\#volatile$ also

[7] Note that Definition 2.11 is not the original definition by Oikarinen and Janhunen (2006) but a more restricted and simpler definition by Gebser et al. (2011a).

contains the parameter t, but here the rules which contain t are removed from the set of LP rules after each iteration.
Formally, one is interested in finding an answer set for a Logic Program

$$R[t] = B \cup \bigcup_{1 \leq j \leq t} P[j] \cup Q[t] \tag{2.42}$$

for some $t \geq 1$ where B represents the $\#base$ part, P represents the $\#cumulative$ part and Q represents the $\#volatile$ part. Incremental ASP solving is realized with *iclingo* (Gebser et al., 2008) and *oclingo* (Gebser et al., 2011a) by incremental grounding. As an auxiliary definition to relate the module theory to the idea of incremental ASP solving, consider Definition 2.13 which describes how a set of rules P is "projected" to a set of atoms X.

Definition 2.13 (Projecting rules onto atoms (Gebser et al., 2008)) *We define for a program P and a set X of atoms the set $P|_X$ as*

$$\begin{aligned} P|_X = \{ & head(r) \leftarrow body(r)^+ \cup L | \\ & r \in P \wedge body(r)^+ \subseteq X \wedge L = \{not\ c | c \in (body(r)^- \cap X)\}\} \end{aligned}$$

As an example consider the following Logic Program:

$$\begin{aligned} P = & \\ & p. \\ & q \leftarrow notp. \\ & v \leftarrow w. \\ & x \leftarrow y. \\ & y \leftarrow not\ q, not\ w. \\ & \leftarrow v. \end{aligned}$$

Let $X = \{p, q\}$, then we have the following "projected" set of rules:

$$\begin{aligned} P|_X = & \\ & p. \\ & q \leftarrow not\ p. \\ & y \leftarrow not\ q. \end{aligned}$$

One can understand $P|_X$ as the set of rules of a LP P where the positive bodies are "compatible" with X and the set of negative body atoms of each rule is truncated, such that only these atoms which are also in X remain. Definition 2.13 allows one to relate non-ground Logic Programs to ground modules. This is described in Definition 2.14.

Definition 2.14 (Relating non-ground LPs to ground modules (Gebser et al., 2008)) *Let P be a non-ground program over a set of predicates $\mathcal{A}$, let $grd(\mathcal{A})$ be the set of atoms occurring in the grounded Logic Program $grd(P)$ and let $I \subseteq grd(\mathcal{A})$ be a set of atoms. Then we define $\mathbb{P}(I)$ as the module*

$$\mathbb{P}(I) = \langle grd(P)|_Y, I, head(grd(P)|_X) \rangle$$

where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.

The Logic Program $P(\mathbb{P}(I))$ is the projection of $grd(P)$ onto inputs and head atoms of $grd(P)$. The output $O(\mathbb{P}(I))$ is the set of head atoms of the Logic Program $grd(P)|_{I \cup head(grd(P))}$

Example for Iterative ASP Solving

As an example for iterative ASP solving consider the following Logic Program (2.43).

$$\begin{aligned}
&\#base.\\
&p(1).\\
&p(2).\\
&\#cumulative\ \ t.\\
&q(t) \leftarrow p(t).\\
&\#volatile\ \ t.\\
&v(t) \leftarrow p(t).\\
&\quad\ \ \leftarrow v(t).
\end{aligned} \tag{2.43}$$

The first grounded LP with the slice for $t = 1$ is (2.44)

$$\begin{aligned}
&p(1).\\
&p(2).\\
&q(1) \leftarrow p(1).\\
&v(1) \leftarrow p(1).\\
&\qquad\ \leftarrow v(1).
\end{aligned} \tag{2.44}$$

Due to the integrity constraint $\leftarrow v(1)$ the LP does not have a Stable Model. Therefore another slice is added to the LP, which results in (2.45)

$$\begin{aligned}
&p(1).\\
&p(2).\\
&q(1) \leftarrow p(1).\\
&q(2) \leftarrow p(2).\\
&v(2) \leftarrow p(2).\\
&\qquad\ \leftarrow v(2).
\end{aligned} \tag{2.45}$$

Note that the LP rules $v(1) \leftarrow p(1)$ and $\leftarrow v(1)$ are not part of the Logic Program anymore, since $v(t) \leftarrow p(t)$ and $\leftarrow v(t)$ appear in the $\#volatile$ part of the non-grounded LP and are therefore discarded in the second iteration. As in (2.44), the LP does not have a Stable Model.
The third iteration produces (2.46)

$$\begin{aligned}
&p(1).\\
&p(2).\\
&q(1) \leftarrow p(1).\\
&q(2) \leftarrow p(2).\\
&q(3) \leftarrow p(3).\\
&v(3) \leftarrow p(3).\\
&\quad\;\; \leftarrow v(3).
\end{aligned} \tag{2.46}$$

Now, since $p(3)$ is not part of the Stable Model $v(3)$ is also not part of the Stable Model and the integrity constraint $\leftarrow v(3)$ does not prevent the LP from having Stable Models. The solution is $\{p(1), p(2), q(1), q(2), v(3)\}$.
Iterative ASP solving is closely related to action theory and action planning in the sense that each iteration of the LP grounding increments the planning horizon by one step. The planning is finished when the planning horizon is broad enough to achieve the stated goal. Note that the online ASP solver *oclingo* (Gebser et al., 2011a) which we are using in our online implementation also supports incremental grounding.

2.2.9. Incremental Online ASP Solving

Online ASP solving as implemented in the solver *oclingo* (Gebser et al., 2011a) is an extension to incremental ASP solving which is also based on the module theory (Oikarinen and Janhunen, 2006). On the syntactic level, a keyword $\#external$ is used to define a set of atoms as external wrt. an incremental Logic Program. If such an atom is received by the ASP solver, then it adds this atom as a new module to the Logic Program and generates new Answer Sets. We denote external atoms by I_P. To understand the semantics of online ASP solving we first describe *online progressions* as defined in (Gebser et al., 2011a).

Definition 2.15 (Online Progression (Gebser et al., 2011a)) *An online progression* $\langle E_i[e_i], F_i[f_i]\rangle_{i\geq 1}$ *is a sequence of pairs of Logic Programs* E_i, F_i *with associated positive integers* e_i, f_i.

Informally, E_i refer to *events* and F_i refer to *inquiries*. Syntactically, the input syntax of *oclingo* represents an online progression with statements of the form

$$\begin{aligned} \#step\ j \\ a_1(t_1), \ldots, a_n(t_n) \\ \#endstep \end{aligned}$$

In this work we only use events E_i in form of Logic Programming facts $a_1(t), \ldots, a_n(t)$ and no inquiries, i.e. $F_i = \emptyset$. In particular, the above statement represents an i-th online progression $\langle \{a_1(t_1), \ldots, a_n(t_n)\}, \emptyset \rangle$, i.e. $E_i = \{a_1(t_1), \ldots, a_n(t_n)\}$.
Online progression statements are sent to the online solver while the main loop is running, and each reception of an online progression triggers a new grounding and solving process. For details concerning the particular interleaving of grounding and solving we refer to Algorithm 1 in (Gebser et al., 2011a). We are now ready to define modularity of online progressions.

Definition 2.16 (Modularity of Online Progressions (Gebser et al., 2011a)) *We define an online progression* $(E_i[e_i], F_i[f_i])_{i \geq 1}$ *as modular wrt. an incremental LP* $\langle B, P[t], Q[t] \rangle$ *if the following modules are defined for all* $j, k \geq 1$ *such that* $e_1, \ldots, e_j, f_j \leq k$.

1. $\mathbb{P}_0 = \mathbb{B}(I_B)$
2. $\mathbb{P}_n = \mathbb{P}_{n-1} \sqcup \mathbb{P}[t/n](O(\mathbb{P}_{n-1}) \cup I_{P[t/n]})$
3. $\mathbb{E}_0 = \langle \emptyset, \emptyset, \emptyset \rangle$
4. $\mathbb{E}_n = \mathbb{E}_{n-1} \sqcup \mathbb{E}_n[e_n](O(\mathbb{P}_{e_n}) \cup O(\mathbb{E}_{n-1}) \cup I_{E_n[e_n]})$
5. $\mathbb{R}_j, k = \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup \mathbb{Q}[t/k](O(\mathbb{P}_k) \cup I_{Q[t/k]}) \sqcup \mathbb{F}_j[f_j](O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]})$

Here, $\mathbb{R}_{j,k}$ is called the *k-expanded* Logic Program of $\langle E_i[e_i], F_i[f_i] \rangle_{1 \leq i \leq j}\ wrt.\ \langle B, P[t], Q[t] \rangle$. For details we refer to (Gebser et al., 2011a, 2012a).

2.3. Modal Logic and the Possible Worlds Model of Knowledge

A *Modal Logic* (e.g. (Blackburn et al., 2001)) is concerned with propositions that depend on modalities. The most common Modal Logic involves the modalities *possibly* (denoted by $\Diamond$) and *necessarily* (denoted by $\Box$). Hintikka (1962) provided a semantics for this Modal Logic which is based on *possible worlds*. Kripke (1963) formulated a deductive

system which allowed him to prove the completeness theorems of Modal Logic.[8] In the following we provide a brief description of Kripke's Semantics for the single-agent case. For a comprehensive study see e.g. (Blackburn et al., 2001) or (Fagin et al., 1995).
A *Kripke Structure* M is triple $\langle W, \pi, R \rangle$ where W is a set of possible worlds, $\pi : \Phi \times W \rightarrow \{true, false\}$ is an interpretation that assigns truth values to formulae **Φ** and R is an accessibility relation between worlds.
The set **Φ** represents first-order formulae augmented with a modal operator □ that denotes knowledge. For instance, to state that an agent knows φ one writes $\Box\varphi$. Intuitively, an agent knows φ if it knows φ in all possible worlds. To denote that an agent knows φ in a particular world w wrt. a Kripke Structure M one writes $\langle M, w \rangle \models \varphi$. The operator $\models$ is defined as follows:

$$\begin{array}{ll} \langle M, w \rangle \models \varphi & \text{iff } \pi(\varphi, s) = true \\ \langle M, w \rangle \models \varphi \wedge \varphi' & \text{iff } \langle M, w \rangle \models \varphi \wedge \langle M, w \rangle \models \varphi' \\ \langle M, w \rangle \models \neg\varphi & \text{iff } \langle M, w \rangle \not\models \varphi \\ \langle M, w \rangle \models \Box\varphi & \text{iff } \forall w' \in W : (\langle w, w' \rangle \in R \Rightarrow \langle M, w' \rangle \models \varphi) \end{array} \tag{2.47}$$

The diamond operator is defined as follows:

$$\Diamond\varphi := \neg\Box\neg\varphi \tag{2.48}$$

In Epistemic Modal Logic, an operator symbol K is commonly used to syntactically replace □ and to denote that a formula is known to hold.
There are different modal axiom schemata which define an epistemic system of a Modal Logic. The schemata are related to so-called *frame conditions*[9] concerning the accessibility relation R: if a frame condition holds for R, then the corresponding axiom schema holds in the particular epistemic system.
Table 2.1 illustrates the most common schemata and their respective frame conditions. Axiom K is called the *distribution axiom* and constitutes a minimal Modal Logic. T is called the *reflexivity axiom*. If it is contained in a Modal Logic, then intuitively all that is known to be true is indeed true. That is, such a Modal Logic constitutes a theory of *knowledge*, not about *belief*. 4 is the *positive introspection* axiom: if an agent knows φ, then it knows that it knows φ. Similarly, 5 reflects *negative introspection*: If an agent does not know that φ, then it knows that it does not know φ.[10] B states that if φ holds, then the agent knows that φ is possible, i.e. it knows that it does not know that not φ holds.
Common Modal Logic systems are enlisted in Table 2.2. K is the minimal Modal Logic.

[8]For a detailed survey on the history of Modal Logic we refer to (Goldblatt, 2003).
[9]Frame conditions in Modal Logic are not to be confused with frame axioms used for circumscription as described in Section 2.1.3.
[10]Axioms 4 and 5 are subject to a fundamental debate in philosophy and many authors (e.g. Williamson (2002)) argue against them.

Label	Axiom	Frame Condition	
K	$\Box(\varphi \to \varphi')$ $\to (\Box\varphi \to \Box\varphi')$	—	—
T	$\Box\varphi \to \varphi$	$\langle w, w\rangle \in R$	reflexive
4	$\Box\Box\varphi \to \Box\varphi$	$\{\langle w, v\rangle, \langle v, u\rangle\} \subseteq R \Rightarrow \langle w, u\rangle \in R$	transitive
5	$\neg\Box\varphi \to \Box\neg\Box\varphi$	$\{\langle w, v\rangle, \langle w, u\rangle\} \subseteq R \Rightarrow \langle v, u\rangle \in R$	euclidean
B	$\varphi \to \Box\Diamond\varphi$	$\langle w, v\rangle \in R \Rightarrow \langle v, w\rangle \in R$	symmetric

Table 2.1.: Common modal axiom schemata and their corresponding frame conditions

Label	Axioms
K	K
KT	K, T
$S4$	$K, T, 4$
$S5$	$K, T, 4, 5, B$

Table 2.2.: Common Modal Logic systems and their axiomatization

KT is a Modal Logic which requires that an agent only knows facts that are true. $S4$ features positive introspection and $S5$ uses also negative introspection.
Most epistemic action theories like $\mathcal{A}_k$ (Son and Baral, 2001), DECKT (Patkos and Plexousakis, 2009) and also the h-approximation consider an equivalent of KT, in the sense that they do not model introspection. The Dynamic Epistmic Logic by van Ditmarsch et al. (2007) and $\mathcal{AOL}$ by Lakemeyer and Levesque (1998) are examples for more expressive formalisms which use introspection. However, in their non-restricted form, they are also undecidable due to the infinite number of state variables which emerge from unlimited nested introspection.

2.4. Epistemic Action Theory: A Survey

In the following we present a survey concerning the state of the art in epistemic action theory. To this end we distinguish theories into four categories.

- *Theories based on a $\mathcal{PWS}$.* This is the traditional approach, following the work by (Hintikka, 1962; Kripke, 1963; Moore, 1985). Model-theoretic $\mathcal{PWS}$-based approaches (e.g. (Scherl and Levesque, 2003)) typically employ Kripke's accessibility relation to model knowledge and operational approaches use multisets of fluents (e.g. (Lobo et al., 2001)).
- *Theories based on disjunctive state-representations* (e.g. (Patkos and Plexousakis, 2009)). Their expressiveness and inference capabilities are comparable to $\mathcal{PWS}$

approaches, but corresponding implementations can be more efficient in practice (To, 2011).

- *Approximations of $\mathcal{PWS}$* (e.g. (Son and Baral, 2001)). These are less expressive but have better computational complexity properties than $\mathcal{PWS}$.

- *Theories that rely on explicit formulation of knowledge-level effects* (e.g. (Petrick and Bacchus, 2004)). Theories of this category can be understood as approximations too, but it is required to carefully specify knowledge-level effects of actions in an epistemically accurate manner. For instance, it has to be explicitly modeled that if the condition of an action is unknown, then its effect is unknown.

We investigate existing approaches of each type and compare their computational properties, expressiveness and inference capabilities. We are primarily interested in the following features which are summarized in Table 2.3.

1. *Number of state variables (exponential or linear) and computational complexity.* The number of state variables is a major factor that determines the computational complexity of an action formalisms. For instance, consider the action language $\mathcal{A}_k$ (Son and Baral, 2001): the plan existence problem is Σ_2^P-complete for an exponential number of state variables and NP-complete for the 0-approximation where the number of state variables is linear wrt. the domain size (Baral et al., 2000).[11]

2. *Elaboration tolerant postdiction.* Postdiction is an inference-pattern which is required to model diagnosis tasks like abnormality detection. We emphasize the need for *elaboration tolerant* (McCarthy, 1998) formalisms that capture postdiction. For example, some action theories (e.g. (Tu et al., 2007)) support so-called Static Causal Laws (SCL). SCL are *if-then* constructs that can be used for an ad-hoc implementation (by explicit encoding) of postdiction. Example 2.1 shows that this approach is not elaboration tolerant.

[11] Under the assumption that plans are polynomial in size.

Example 2.1 Elaboration Tolerance and Static Causal Laws
A robot can execute an action $\texttt{drive}_d$ to reach a room through a door d. A fluent $\texttt{in}$ denotes that it is in the room, and a fluent $\texttt{open}_d$ denotes that the door d is open. An auxiliary fluent $\texttt{did_drive}_d$ represents that $\texttt{drive}$ has been executed. A manually encoded SCL postdicts that if the robot is $\texttt{in}$ the destination room after driving the door must be $\texttt{open}$: "If $\texttt{did_drive}_d$ and $\texttt{in}$ then $\texttt{open}_d$". The robot has a location sensor to determine whether it is in the room. The sensor is activated with an action $\texttt{sense_in}$. Consider a scene where the robot initially does not know whether the door is open or closed. It executes first $\texttt{drive}_1$ and then $\texttt{sense_in}$. Here $\mathcal{A}_k^c$ correctly generates knowledge that $\texttt{open}_1$ holds if the robot indeed arrived in the room. Now consider an elaboration of the problem with two doors, i.e. $d \in \{1, 2\}$. The robot could first try to drive through door 1 ($\texttt{drive}_1$), then drive through door 2 ($\texttt{drive}_2$) and then sense its location ($\texttt{sense_in}$). Here, $\texttt{did_drive}_1$ becomes true after $\texttt{drive}_1$, regardless of the actual open-state of door 1. Therefore, if door 1 is closed and the robot actually passed door 2 to get into the room, then the SCL would produce the wrong conclusion that door 1 is open. Hence, the workaround is not elaboration tolerant.

3. *Concurrent acting and sensing.*
   Real-world domains often demand to model actions which change the world and concurrently sense a property of the world.[12] For instance, pulling the trigger of a gun causes the shooter to know whether the gun was loaded and at the same time has the physical effect of the impact of the bullet (see Example 7.1). Another more practical example is a sensor which consumes energy while the sensing happens.

4. *Temporal knowledge dimension.*
   Reasoning about past (or future) facts opens up a new range of problems, for instance in narrative interpretation or forensic reasoning. Witnesses may give evidence about facts in the past; or knowledge about the occurrence of an event may be acquired at a later time point. Example 7.1 illustrates that a temporal knowledge dimension is also required to model actions which sense and concurrently affect the same fluent.

5. *Implementation.*
   A major goal of our research is to apply $\mathcal{HPX}$ in real robots and other applications which require an implementation. An implementation also serves as a proof-of concept of a formalism.

[12] In fact, the Uncertainty Principle by Heisenberg (1927) states that – at least on the quantum physical level – sensing is actually *impossible* without concurrent physical side-effects.

6. *Soundness / completeness results.*
   Many formalism are defined in "isolation" wrt. other approaches, i.e. the relation to other formalisms is not investigated formally by performing soundness or completeness proofs. Without such proofs, the relation between different action theories is unclear and conditions under which a formalism is equally expressive as another formalism can not be identified.

2.4.1. $\mathcal{PWS}$-based Epistemic Action Theories

The most popular approach to represent knowledge is a non-approximated $\mathcal{PWS}$: knowledge is represented by an exponential number of possible worlds. The idea behind this approach stems from the work concerning Epistemic Modal Logic by Hintikka (1962), Kripke (1963) and Moore (1985) which we described in Section 2.3. That is, knowledge states are modeled either with an modal-logical *accessibility relation* (e.g. (Scherl and Levesque, 2003)) or with multisets of fluents in mathematical logic (e.g. (Son and Baral, 2001)). These approaches support postdictive reasoning in an elaboration tolerant manner, but they have the disadvantage that modeling an agent's knowledge state requires an exponential number of possible worlds and hence an exponential number of state variables.
For instance, Lobo et al. (2001) use both mathematical logic and epistemic logic programming to formulate a $\mathcal{PWS}$ based epistemic extension to the action language $\mathcal{A}$. Another $\mathcal{PWS}$ based semantics for $\mathcal{A}$ is defined for the action language $\mathcal{A}_k$ by Son and Baral (2001). The semantics is sound and complete wrt. the approach by Scherl and Levesque (2003) and the approach by (Lobo et al., 2001). The $\mathcal{PWS}$-based semantics for $\mathcal{A}_k$ has not been implemented and does not feature concurrent actions in general. A special concern on $\mathcal{A}_k$ is given in Section 3.4, where we extend its semantics towards the temporal dimension of knowledge.
Scherl and Levesque (2003) provide an epistemic extension and a solution to the frame problem for the Situation Calculus (McCarthy, 1963) using an accessibility relation. A temporal knowledge dimension does not exist and concurrency are not supported. To the best of our knowledge there exists no implementation of their version of the epistemic SC. Son and Baral (2001) showed that under certain assumptions Scherl and Levesque's approach is sound and complete wrt. the $\mathcal{PWS}$-based semantics for the action language $\mathcal{A}_k$.
Lakemeyer and Levesque (1998) combine knowledge and action in the logic $\mathcal{AOL}$ in the context of the situation calculus and the logic of *only knowing* (Levesque, 1990). Their approach considers introspection, i.e. knowledge about knowledge, and for this reason the number of state variables is in infinite. A temporal dimension of knowledge and concurrency is not considered. To the best of our knowledge, the theory has not been implemented and the relation to other epistemic action calculi has not been investigated

Features of epistemic action theories	**# State variables**	**Elaboration tolerant postdiction**	**Temporal knowledge dimension**	**Concurrent acting and sensing**	**Implementation**	**Soundness / completeness results**
$\mathcal{PWS}$-based approaches						
Extension to $\mathcal{A}$ (Lobo et al., 2001)	EXP	✓	-	-	-	✓
$\mathcal{A}_k(\mathcal{PWS})$ (Son and Baral, 2001)	EXP	✓	-	-	-	✓
Epistemic SC (Scherl and Levesque, 2003)	EXP	✓	-	-	-	✓
$\mathcal{AOL}$ (Lakemeyer and Levesque, 1998)	∞	✓	-	-	-	-
Epistemic FC (Thielscher, 2000)	EXP	✓	-	-	(✓)	✓
CFF (Hoffmann and Brafman, 2005)	EXP	✓	-	-	✓	-
EFEC (Ma et al., 2013)	EXP	✓	✓	✓	✓	-
Approach by (Vlaeminck et al., 2012)	EXP	✓	✓	-	-	-
Disjunctive state approaches						
DECKT (Patkos and Plexousakis, 2009)	EXP	✓	-	(✓)	✓	✓
PrAO (To, 2012)	EXP	✓	-	-	✓	-
Approximations of $\mathcal{PWS}$						
h-approximation ($\mathcal{HPX}$)	**LIN**	✓	✓	✓	✓	✓
Epistemic SC (Demolombe and del Pilar Pozos Parra, 2000)	LIN	-	-	-	-	-
Epistemic SC (Liu and Levesque, 2005)	LIN	-	-	-	-	-
$\mathcal{A}_k^c$(0-approximation) (Tu et al., 2007)	LIN	-	-	(✓)	✓	✓
Approaches requiring explicit knowledge-level effects						
PKS (Petrick and Bacchus, 2004)	LIN	-	-	(✓)	✓	-
FLUX (Thielscher, 2005)	LIN	-	-	(✓)	✓	(✓)
INDIGOLOG (de Giacomo and Levesque, 1998)	LIN	-	-	(✓)	✓	-

Table 2.3.: Survey on epistemic action theories

formally.

The Fluent Calculus (FC) (Hölldobler and Schneeberger, 1990; Thielscher, 1998) was extended to consider knowledge in (Thielscher, 2000). It uses an accessibility relation similar to (Scherl and Levesque, 2003) and hence requires an exponential number of state variables. The epistemic FC does not consider a temporal knowledge dimension. There exists an extension which covers concurrency (Thielscher, 2001), but this extension does not consider knowledge and sensing. Kahramanogullari and Thielscher (2003) provide a formal investigation concerning the relation between FC and the epistemic extension of $\mathcal{A}$ by Lobo et al. (2001). There exists a Prolog implementation of the epistemic Fluent Calculus called FLUX (Thielscher, 2005), but its semantics differs from the original epistemic FC in that it does not employ an accessibility relation. For this reason we report about FLUX separately.

In addition to logic-based action-theoretic approaches there exist several PDDL-based (McDermott et al., 1998) planners that deal with incomplete knowledge. These planners achieve high performance via practical optimizations such as BDDs (Bertoli et al., 2001) or heuristics-driven search algorithms like those used for the ContingentFF (CFF) planner (Hoffmann and Brafman, 2005). However, their underlying semantics is typically based on a $\mathcal{PWS}$. For instance, the semantics of CFF is based on sequential STRIPS (Fikes and Nilsson, 1972) and adds the $\mathcal{PWS}$-approach to model sensing and knowledge. Despite its efficiency in relatively small domains, the exponential number of state variables causes an extreme phase transition if the domain size exceeds a certain threshold.[13] Postdiction is possible and elaboration tolerant, but concurrent acting and sensing is not supported in CFF. A formal investigation of the relation between CFF and other formalisms is also not provided.

To the best of our knowledge, there are only two approaches that consider the temporal dimension of knowledge. Ma et al. (2013) proposed an epistemic extension to the Event Calculus which considers *possible world histories* instead of possible worlds. It has elaboration tolerant support for postdiction and knowledge about past and future can be represented. Concurrent acting and sensing is possible with EFEC, and an implementation exists as Answer Set Programming.[14]

Another $\mathcal{PWS}$-based first-order logical framework to model a temporal dimension of knowledge is provided by Vlaeminck et al. (2012). The framework employs first-order-reasoning such that it allows for elaboration tolerant postdiction and the projection problem is solvable in polynomial time. However, the authors do not provide a practical implementation and evaluation of their method. The key feature of their approach is that the first-order reasoning is approximated, such that the plan-existence problem is in NP

[13] For example, the RING problem (Hoffmann and Brafman, 2005) where an agent must move through n rooms and close the windows in these rooms demands 1.5s for 4 rooms, 480s for 5 rooms and produces a timeout $>$ 3600s for 6 rooms with CFF on a 2 Ghz i5 machine with 6GB RAM.

[14] http://www.ucl.ac.uk/infostudies/efec/ (accessed on 17th July 2013)

despite the exponential number of state variables of the actual action formalism.[15] The framework has not been implemented and the relation to other action calculi has not been formally investigated.

2.4.2. Theories with a Disjunctive Knowledge State Representation

There are alternatives to a $\mathcal{PWS}$-based knowledge representations which use a disjunctive knowledge representation. One example is DECKT (Patkos and Plexousakis, 2009) – an epistemic extension to the Event Calculus. DECKT relies on so-called *Hidden Causal Dependencies* which are modeled by reified first-order predicates. For instance $Knows(\neg f_1 \vee f_2, T)$ represents that at a time T an agent has knowledge about the causal dependency "if f_1 holds then f_2 must also hold". In general, disjunctive approaches require only a linear number of states to model an agent's knowledge but the number of variables per state is up to exponential. An implementation of DECKT is presented in (Patkos and Plexousakis, 2012) but the implementation is not capable of action planning. DECKT is sound and complete wrt. BDECKT (Patkos and Plexousakis, 2009), a $\mathcal{PWS}$-based version of the Event Calculus. Concurrent acting and sensing is in principle possible but fails if an action senses the value of a fluent which is modified at the same time, as demonstrated in our Yale Shooting Scenario in Example 7.1.
Another approach is presented by To (2012). The author implements a performance-wise very successful general planning framework called PrAO which is based on so-called *minimalDNS* representations. Concurrency is not supported and a formal comparison with other epistemic action calculi is not presented.

2.4.3. Approximate Epistemic Action Theories

Approximate theories are usually derived from a $\mathcal{PWS}$ based formalization. Approximations can have simpler state representations and hence a lower computational complexity, but postdiction is not naively supported with existing approaches. Demolombe and del Pilar Pozos Parra (2000) provide an approximate epistemic extension to the Situation Calculus (SC) which is based on special knowledge fluents. Their approach involves simpler frame axioms compared to the $\mathcal{PWS}$ based approach by Scherl and Levesque (2003), such that an implementation would be tractable. However, postdiction is not possible with this approach. A temporal knowledge dimension and concurrency is also not supported. Liu and Levesque (2005) present another epistemic extension of the Situation calculus. Their approach to approximate $\mathcal{PWS}$ is based on so-called *local action effects*. The intuition behind local action effects is that all conditions of an action

[15]The authors only provide a complexity result that the projection problem is polynomial, but it follows directly from the definition of non-deterministic Turing machines that plan-existence is in NP.

must be known before execution. This implies that postdiction – in the sense of inferring the conditions of an action by observing its effect – is not required. The approach by (Liu and Levesque, 2005) is claimed to coincide with the 0-approximation of Son and Baral (2001) if formulae are restricted to be propositional. However, we were unable to locate a formal proof.
The *0-approximation* by Son and Baral (2001) does not consider an exponential number of possible worlds, but instead one single approximate world which is the intersection of all possible worlds. This requires only a linear number of state variables to model the knowledge state of an agent, instead of an exponential number with $\mathcal{PWS}$ based approaches. The plan-existence problem for $\mathcal{A}_k$ is NP-complete (Baral et al., 2000), and postdiction is not supported.[16] Tu et al. (2007) introduce $\mathcal{A}_k^c$ and add Static Causal Laws (SCLs) to the 0-approximated $\mathcal{A}_k$. In Example (ex:AckNotElTolerant) we demonstrate that SCL allow only for a non-elaboration tolerant form of Postdiction.

2.4.4. Epistemic Action Theories with Explicit Knowledge-Level Effects

Approaches like the PKS planner (Petrick and Bacchus, 2004), the FLUX system (Thielscher, 2005) or INDIGOLOG (de Giacomo and Levesque, 1998) require to model the knowledge-level effects of an agent explicitly in the action specification. These are able to deal with incomplete knowledge, but knowledge-level effects of actions have to be defined manually for each action.
An example is given wrt. the PKS planner by Petrick and Bacchus (2004): consider a *dial* action that is supposed to open a safe if the dialed combination is correct. If it is known that the safe is initially closed, and if it is unknown whether the dialed combination is correct, then obviously knowledge about the closed-ness of the safe is lost after dialing, because the dialed combination may or may not be correct and the safe may or may not open. In PKS, the epistemic effect of knowledge loss must be explicitly stated. In consequence, epistemic accuracy of the specification is not guaranteed because the definition of knowledge-level effects is left to the domain designer. Postdiction can be implemented in a similar manner, but this also has to be done manually in the action specification which is not elaboration-tolerant. PKS is based on the Epistemic Situation Calculus by (Scherl and Levesque, 2003), but a formal soundness proof wrt. this or other formalisms is not presented. There exists an implementation of PKS which is particularly efficient if many functional fluents are used.
FLUX (Thielscher, 2005) can be understood as an implementation of the (epistemic) Fluent Calculus in Prolog. However, there is a fundamental difference between the epistemological reasoning machineries of both calculi: Unlike the original epistemic

[16] Son and Baral (2001) present other approximations (e.g. 1-approximation or ω-approximation) in the same paper as well, but these also do not support postdiction.

Fluent Calculus defined in (Thielscher, 2000), FLUX does not employ an accessibility relation but instead an explicit knowledge-predicate. As a result, knowledge-level effects of actions (such as postdiction) have to be specified manually which makes a formal investigation of the relation between this limited form of epistemic reasoning and the epistemic FC difficult; soundness or completeness results are not provided. The manual specification of knowledge-level effects make FLUX also elaboration-intolerant.
INDIGOLOG (de Giacomo and Levesque, 1998) is a high-level Cognitive Robotics control framework that supports sensing and incomplete knowledge. It is based on an operational semantics and implemented in Prolog. In the implementation, sensing is modeled in a simplified manner and an accessibility relation is not employed. Though work concerning epistemic accuracy of INDOGOLOG has been conducted in (Sardina et al., 2004), we were unable to find actual soundness or completeness results for INDIGOLOG's semantics wrt. other epistemic action theories like (Scherl and Levesque, 2003).
Concurrent acting and sensing is possible with all enlisted approaches (PKS, FLUX and INDIGOLOG) but fails for actions which sense and concurrently change the same fluent's value, as described in our extended version of the Yale Shooting Problem (Example 7.1).

3 $\mathcal{HPX}$: The h-Approximation

This chapter describes the operational semantics of the h-approximation. We first describe the syntactic elements of the PDDL-like input language and state how these map to a set-theoretic formalization of $\mathcal{HPX}$ (Section 3.1). The formalization is based on so-called *h-states* which represent the knowledge state of an agent. An h-state considers a set of pairs of fluent literals and time points which represent the *knowledge history* of an agent. In addition, a set of pairs of actions and time points represent the *action history*.
Section 3.2 describes how state transitions are modeled. We define a transition function which maps an h-state and a set of actions to a set of h-states. State transitions involve a re-evaluation step which iteratively refines the knowledge history of an agent by considering sensing results and the occurrence of actions. Based on the basic transition function for single actions, we formalize concurrent conditional plans (CCP) and define an extended transition function which maps a CCP and an h-state to a set of h-states. We illustrate the theory with a running minimal example of a robot trying to enter a room by driving through a door.
In Section 3.3 we discuss the computational complexity of $\mathcal{HPX}$. Theorem 3.1 states that the plan-existence problem for $\mathcal{HPX}$ is in NP.
We conclude the chapter with Section 3.4, which demonstrates soundness of $\mathcal{HPX}$ wrt. traditional epistemic action theories based on a possible-worlds semantics. To this end we define an extended temporal semantics for the action language $\mathcal{A}_k$(Son and Baral, 2001), which is $\mathcal{PWS}$-based and capable of representing temporal knowledge.

3.1. Domain Specification and Syntax

Throughout this thesis we use the following notational conventions: We use the symbol a to denote *actions*, ep for *effect proposition*, n and t for *time* (or step), b for *branch*, and f for *fluent*. l denotes *fluent literals* of the form f or $\neg f$. $\bar{l}$ denotes the complement of l

and $|l|$ is used to "positify" a literal, i.e. $|\neg f| = f$ and $|f| = f$.
With these conventions we describe the syntax of our PDDL dialect to specify planning domains denoted $\mathcal{D}$. $\mathcal{D}$ consists of the language elements (3.1a) – (3.1f) as follows:

$$\texttt{(:init (and } l_1^{init} \ldots l_{n_{in}}^{init}\texttt{))} \tag{3.1a}$$

$$\texttt{(oneof } l_1^{isc} \ldots l_n^{isc}\texttt{)} \tag{3.1b}$$

$$\texttt{(:action } a \texttt{ :effect if (and } l_1^c \ldots l_{n_c}^c\texttt{) then } l^e\texttt{)} \tag{3.1c}$$

$$\texttt{(:action } a \texttt{ :observe } f^s\texttt{)} \tag{3.1d}$$

$$\texttt{(:action } a \texttt{ :executable (and } l_1^{ex} \ldots l_{n_{ex}}^{ex}\texttt{))} \tag{3.1e}$$

$$\texttt{(:goal}\,\mathrm{type}\,\texttt{(and } l_1^g \ldots l_{n_g}^g\texttt{))} \tag{3.1f}$$

(3.1a). A set of *value propositions* ($\mathcal{VP}$) denote initial facts. Formally, for an expression (3.1a) $\mathcal{VP} = \{l_1^{init}, \ldots, l_{n_{in}}^{init}\}$ is a set of fluent literals.

(3.1b). A set of initial state constraints ($\mathcal{ISC}$) denotes exclusive-or knowledge about the initial state. Formally, an ISC is a set of literals, e.g. for (3.1b) we have one ISC $\mathcal{C} = \{l_1^{isc}, \ldots, l_n^{isc}\}$ and $\mathcal{C} \in \mathcal{ISC}$.

(3.1c). A set of *effect propositions* (EP) of an action a (denoted $\mathcal{EP}^a$) represents conditional action effects . We call $l_1^c \ldots l_{n_c}^c$ condition literals and l^e effect literals. $c(ep)$ denotes the set of condition literals and $e(ep)$ denotes the effect literal of an effect proposition ep. Formally, an EP is the pair $\langle c(ep), e(ep)\rangle$. An action has a finite number of EPs and we write $ep_i(a)$ to denote the i-th effect proposition of an action a.

(3.1d). A *knowledge proposition* of an action a, denoted $\mathcal{KP}^a$, represents that an action senses a fluent f^s, i.e. for (3.1d) we have $\mathcal{KP}^a = f^s$.

(3.1e). *Executability conditions* ($\mathcal{EXC}^a$) denote what an agent must know in order to execute an action a. Formally, an executability condition is a set of literals, i.e. for (3.1e) we have $\mathcal{EXC}^a = \{l_1^{ex}, \ldots, l_{n_{ex}}^{ex}\}$

(3.1f). *Goal propositions* ($\mathcal{G}^{strong}$, $\mathcal{G}^{weak}$) denote goals, where $\mathrm{type} \in \{\texttt{weak}, \texttt{strong}\}$. Formally, $\mathcal{G}^{strong}$ is the set of literals specified with $\mathrm{type} = \texttt{strong}$ and $\mathcal{G}^{weak}$ is the set of literals specified with $\mathrm{type} = \texttt{weak}$. Weak goals denote that a plan has to be found which *possibly* achieves the goal. That is, there must be at least one leaf state in the transition tree where the goal is achieved. A strong goal must be achieved in all leaf states, i.e. a plan must *necessarily* achieve a goal. The formal difference between weak and strong goals is discussed in Section 3.2.11.

Note that a set of domain fluents (denoted $\mathcal{F}_{\mathcal{D}}$) is implicitly defined by the domain definition: whenever a fluent literal f or $\neg f$ is involved in one of the language elements (3.1a) – (3.1f), then f is contained in the set of domain-fluents $\mathcal{F}_{\mathcal{D}}$. Respectively, $\mathcal{L}_{\mathcal{D}}$ is the set of domain literals.

3.2. Operational Semantics of the H-Approximation

The semantics is defined in terms of a transition function (3.7) that maps actions and state histories to state histories. To realize postdiction and other epistemic effects, a re-evaluation function $eval$ is applied after each state transition. $eval$ retrospectively considers temporal knowledge and incrementally refines knowledge with inference mechanisms for postdiction, causation and inertia. In addition to the transition function (3.7) for single actions we define an extended transition function (3.18) that maps a *concurrent conditional plan* and a state to a set of states.[1]
Formally, a planning domain $\mathcal{D}$ is a tuple $\langle \mathcal{VP}, \mathcal{ISC}, \mathcal{A}, \mathcal{G} \rangle$ where:

- $\mathcal{VP}$ is a set of value propositions (3.1a)
- $\mathcal{ISC}$ is a set of initial state constraints (3.1b)
- $\mathcal{A}$ is a set of actions. A non-sensing action a is a pair $\langle \mathcal{EP}^a, \mathcal{EXC}^a \rangle$ consisting of a set of effect propositions $\mathcal{EP}^a$ (3.1c) and an executability condition $\mathcal{EXC}^a$ (3.1e). Sensing actions are represented as a tuple $\langle \mathcal{KP}^a, \mathcal{EP}^a, \mathcal{EXC}^a \rangle$ where $\mathcal{KP}^a$ (3.1d) denotes a knowledge proposition.
- $\mathcal{G} = \left\langle \mathcal{G}^{strong}, \mathcal{G}^{weak} \right\rangle$ is a pair of strong and weak goal propositions (3.1f).

3.2.1. Knowledge States with a Temporal Dimension

The $\mathcal{HPX}$ semantics is based on so-called *history-states* (h-states) $\mathfrak{h}$ which are pairs $\langle \boldsymbol{\alpha}, \boldsymbol{\kappa} \rangle$. $\boldsymbol{\alpha}$ denotes the action history and $\boldsymbol{\kappa}$ the knowledge history of $\mathfrak{h}$. Formally, $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ are represented as follows:

- An *action history* $\boldsymbol{\alpha}$ consists of pairs $\langle a, t \rangle$ where a is an action and t is a time step.
- A *knowledge history* $\boldsymbol{\kappa}$ is a set of pairs $\langle l, t \rangle$ where l is a literal and t is a time-step.

[1] Our definition of concurrency is restricted in the sense that all actions are assumed to have the same duration. Therefore we define concurrency only wrt. single actions and not wrt. to other concurrent conditional (sub-)plans.

To handle *concurrency* in a more convenient manner we introduce *effect histories* as an auxiliary instrument derived from action histories.

- An *effect history* $\boldsymbol{\epsilon}$ is a set of pairs $\langle ep, t\rangle$ where ep is an effect proposition and t is a time-step.

The formal definition of effect histories is provided in Definition 3.1.

Definition 3.1 (Effect history $\boldsymbol{\epsilon}$) *Let $\boldsymbol{\alpha} = \{\langle a_1, t_1\rangle, \ldots, \langle a_n, t_n\rangle\}$ be an action history and let $\mathcal{EP}^a$ denote the set of effect proposition of an action a. Then the effect history $\boldsymbol{\epsilon}(\boldsymbol{\alpha})$ of the action history $\boldsymbol{\alpha}$ is given by (3.2).*

$$\boldsymbol{\epsilon}(\boldsymbol{\alpha}) = \{\langle ep, t\rangle \,|\exists \langle a, t\rangle \in \boldsymbol{\alpha} : ep \in \mathcal{EP}^a\} \tag{3.2}$$

For convenience we also write (3.3).

$$\boldsymbol{\epsilon}(\mathfrak{h}) = \boldsymbol{\epsilon}(\boldsymbol{\alpha}(\mathfrak{h})) \tag{3.3}$$

In general, we write $\boldsymbol{\alpha}(\mathfrak{h})$, $\boldsymbol{\kappa}(\mathfrak{h})$ and $\boldsymbol{\epsilon}(\mathfrak{h})$ to denote the action history, knowledge history and effect history of an h-state $\mathfrak{h}$. To simplify notation, we sometimes transfer sub- and superscripts from $\mathfrak{h}$ to $\boldsymbol{\epsilon}$, $\boldsymbol{\kappa}$ and $\boldsymbol{\alpha}$ (if clear from the context). For instance we write $\boldsymbol{\epsilon}_n$ to denote $\boldsymbol{\epsilon}(\mathfrak{h}_n)$.

3.2.2. Initial Knowledge

A particular h-state of a domain description is the *initial state*, described by Definition 3.2.

Definition 3.2 (Initial h-state $\mathfrak{h}_0$) *A state is called the* initial state *(denoted by $\mathfrak{h}_0 = \langle \boldsymbol{\alpha}_0, \boldsymbol{\kappa}_0\rangle$) of a domain $\mathcal{D}$ if and only if*

1. $\boldsymbol{\alpha}_0 = \emptyset$
2. *for every fluent literal l in a value proposition $\mathcal{VP}$: $\langle l, 0\rangle \in \boldsymbol{\kappa}_0$.*
3. *for every initial state constraint $\mathcal{C} \in \mathcal{ISC}$:*

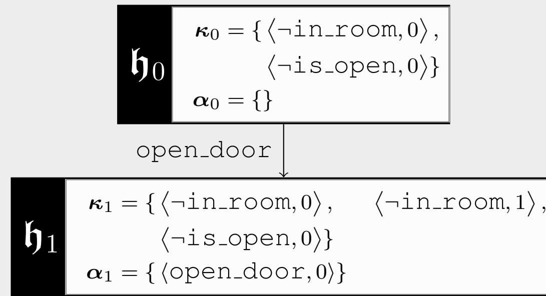
$$\begin{aligned} \forall l \in \mathcal{C} : \quad & \langle l, 0\rangle \in \boldsymbol{\kappa}_0 \Rightarrow (\forall l' \in \mathcal{C} \setminus l : \left\langle \overline{l'}, 0\right\rangle \in \boldsymbol{\kappa}_0) \\ & \wedge (\forall l' \in \mathcal{C} \setminus l : \left\langle \overline{l'}, 0\right\rangle \in \boldsymbol{\kappa}_0) \Rightarrow \langle l, 0\rangle \in \boldsymbol{\kappa}_0 \end{aligned} \tag{3.4}$$

Example 3.1 depicts how an action is applied to transform the initial state $\mathfrak{h}_0$ into a successor state $\mathfrak{h}_1$.

```
(:action open :effect if ¬jammed then is_open)
(:init ¬in_room ¬is_open)
(:goal weak is_open)
```

Listing 3.1: Opening an potentially jammed door

Example 3.1 Action and knowledge history
Consider Listing 3.1 which specifies the problem of driving through a door into a room if it is unknown whether the door is open. The value proposition `(:init ¬is_open)` results in the initial knowledge history $\boldsymbol{\kappa}_0 = \{\langle \neg\texttt{is_open}, 0\rangle\}$ and the action history $\boldsymbol{\alpha}_0 = \emptyset$. Applying action `open_door` on $\mathfrak{h}_0$ causes a transition to $\mathfrak{h}_1$:



Note that $\boldsymbol{\kappa}_1$ does not contain a pair $\langle \texttt{in_open}, 1\rangle$ or $\langle \neg\texttt{is_open}, 1\rangle$: If it is unknown whether there is an abnormality in opening the door, then it is also unknown whether the door is actually open after executing this action.

3.2.3. Knowledge about the Presence (and the Past)

To identify the *present world state* within a knowledge history we define an auxiliary function now (3.5): it returns the number of state transitions that have occurred so far.

$$now(\mathfrak{h}) = \begin{cases} 0 & \text{if } \boldsymbol{\alpha}(\mathfrak{h}) = \emptyset \\ t+1 & \text{if } \exists \langle a, t\rangle \in \boldsymbol{\alpha}(\mathfrak{h}) : \forall \langle a', t'\rangle \in \boldsymbol{\alpha}(\mathfrak{h}) : t' \leq t \end{cases} \tag{3.5}$$

To represent knowledge about the past and the presence, we use an entailment operator $\models$ to define *(a)* whether a literal l is known to hold at the present step (3.6a), or *(b)* whether a pair $\langle l, t\rangle$ is known to hold (3.6b), i.e. whether at the present step l is known to hold at a possibly earlier step t.

$$\mathfrak{h} \models l \Leftrightarrow \langle l, now(\mathfrak{h})\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \tag{3.6a}$$

$$\mathfrak{h} \models \langle l, t\rangle \Leftrightarrow \langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \tag{3.6b}$$

3.2.4. Executability of Actions

Executability conditions (3.1e) are qualifications on the agent's knowledge at the time it executes an action. They reflect what an agent must know in order to execute an action. Executability conditions are formalized in Definition 3.3.

Definition 3.3 (Executability of actions) *Consider an action a with an executability condition $\mathcal{EXC}^a = \{l_1^{ex}, \ldots, l_{n_{ex}}^{ex}\}$. We say that a is executable in an h-state $\mathfrak{h}$ if $\forall l^{ex} \in \mathcal{EXC}^a : \mathfrak{h} \models \langle l^{ex}, now(\mathfrak{h})\rangle$.*

Intuitively, an action is executable if all literals in the executability condition are known to hold at the step the action is executed, i.e. at $now(\mathfrak{h})$.

3.2.5. Sensing, Branching, Transition Function

The transition function Ψ (3.7) adds a set of actions to the action history $\boldsymbol{\alpha}$ and then evaluates the knowledge-level effects of these actions. Ψ considers sensing and maps a set of actions $\boldsymbol{A}$ and a state to a set of states.

$$\Psi(\mathbf{A}, \mathfrak{h}) = \bigcup_{k \in sense(\mathbf{A}^{ex}, \mathfrak{h})} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k \rangle) \tag{3.7}$$

where

- $\boldsymbol{A}^{ex}$ is the subset of actions of $\boldsymbol{A}$ which are executable in $\mathfrak{h}$
- $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(\mathfrak{h}) \cup \{\langle a, t\rangle \,|a \in \mathbf{A}^{ex} \wedge t = now(\mathfrak{h})\}$

The transition function calls two other function, $sense$ and $eval$:

- $eval$ (3.17) is a re-evaluation function which we describe in Section 3.2.8. In brief, $eval$ refines the knowledge-history of an h-state by determining the knowledge-level effects of non-sensing actions using certain inference mechanisms.
- $sense$ adds sensing results to the knowledge history. It is formally defined as follows. Let $t^s = now(\mathfrak{h})$ then:

$$sense(\mathbf{A}, \mathfrak{h}) = \begin{cases} \{\{\langle f^s, t^s\rangle\}, \{\langle \neg f^s, t^s\rangle\}\} & \\ & \text{if } \mathbf{A} \text{ contains exactly one sensing action } a \\ & \text{with a knowledge proposition } \mathcal{KP}^a = f^s \\ & \text{and } \{\langle f^s, t^s\rangle, \langle \neg f^s, t^s\rangle\} \cap \boldsymbol{\kappa}(\mathfrak{h}) = \emptyset \\ \{\emptyset\} & \text{otherwise} \end{cases} \tag{3.8}$$

Intuitively, $sense$ describes that knowledge is added to the original h-state if none of the possible outcomes of the sensing (either f^s or $\neg f^s$) is already known. Note

that the time at which the sensing result holds is the time at which the sensing happens, i.e. the time before the successor-state time: $t^s = now(\mathfrak{h})$. Example 7.1 demonstrates that this is important to model concurrent acting and sensing.

The re-evaluation function $eval$ consists of five inference mechanisms which constitute the re-evaluation process. Before defining these inference mechanisms we need to introduce the auxiliary notion of *intermeditate h-states* which result from a *partial* re-evaluation. Intermediate h-states are used in Examples 3.2 – 3.5 which illustrate the inference mechanisms.

3.2.6. Intermediate h-states

We define *intermediate h-states*, denoted by a "tilde" symbol (e.g. $\tilde{\mathfrak{h}}$) to represent partial state-transitions. That is, if an action a is applied to an h-state $\mathfrak{h}$, then we add the action to the action history but do not necessarily add all effects of the action to the knowledge history. The formal definition of intermediate h-states is as follows:

Definition 3.4 (Intermediate h-states) *Given h-states* $\mathfrak{h}, \tilde{\mathfrak{h}}$ *and a set of actions* $\mathbf{A}$. *We say that*

$$\tilde{\mathfrak{h}} \text{ is an intermediate h-state of } \mathfrak{h} \text{ wrt. } \mathbf{A}$$

if the following holds:

$$\tilde{\mathfrak{h}} = \langle \boldsymbol{\alpha}(\mathfrak{h}) \cup \{\langle a, now(\mathfrak{h})\rangle \,|a \in \mathbf{A}\}, \tilde{\boldsymbol{\kappa}}\rangle \tag{3.9}$$

where $\tilde{\boldsymbol{\kappa}}$ *is called* intermediate knowledge history *with* $\tilde{\boldsymbol{\kappa}} \supseteq \boldsymbol{\kappa}(\mathfrak{h})$.

With the intermediate h-states we are ready to define and to illustrate five individual inference mechanisms which constitute the re-evaluation process. Intuitively, intermediate h-states denote h-states which are not completely re-evaluated. For example, consider an h-state where a sensing result is added, but postdictive conclusions have not been drawn yet, even though this would be possible.

3.2.7. Inference Mechanisms (IM1.–IM.5)

The evaluation function employs the following five *inference mechanisms* (IM):

IM.1 — Forward inertia of knowledge

IM.2 — Backward inertia of knowledge

IM.3 — Causation

IM.4 — Positive postdiction

IM.5 — Negative postdiction

► IM.1 – IM.2: Inertia

To define how knowledge persists, we first formalize inertia (3.10). Intuitively, a literal l is inertial at a step t if no effect proposition can negate l.

$$inertial(l,t,\mathfrak{h}) = \begin{cases} true & \text{if } \forall \langle ep,t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}): \\ & \quad \big(e(ep)=\bar{l}\big) \Rightarrow \big(\exists l^c \in c(ep) : \langle \overline{l^c}, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h})\big) \\ false & \text{otherwise} \end{cases} \quad (3.10)$$

A literal l is inertial at a step t if *(a)* there is no effect proposition such that $\langle ep,t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h})$ and ep has a complementary effect literal $\bar{l}$, or *(b)* there is an EP such that $\langle ep,t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h})$ and ep has a complementary effect literal $\bar{l}$, but ep has at least one condition literal l^c which is known not to hold at t.
Having defined when a fluent is inertial, we can define forward and backward inertia of knowledge: to this end, we state two functions fwd (3.11) and $back$ (3.12) that map an h-state to an h-state. Forward inertia is defined by (3.11).

$$\begin{aligned} &fwd(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{fwd}(\mathfrak{h})\rangle \\ &\text{where} \\ &add_{fwd}(\mathfrak{h}) = \{\langle l,t\rangle \mid \langle l,t-1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \wedge inertial(l,t-1,\mathfrak{h}) \wedge t \leq now(\mathfrak{h})\} \end{aligned} \quad (3.11)$$

Backward inertia is defined by (3.12).

$$\begin{aligned} &back(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{back}(\mathfrak{h})\rangle \\ &\text{where} \\ &add_{back}(\mathfrak{h}) = \{\langle l,t\rangle \mid \langle l,t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \wedge inertial(\bar{l},t,\mathfrak{h}) \wedge t \geq 0\} \end{aligned} \quad (3.12)$$

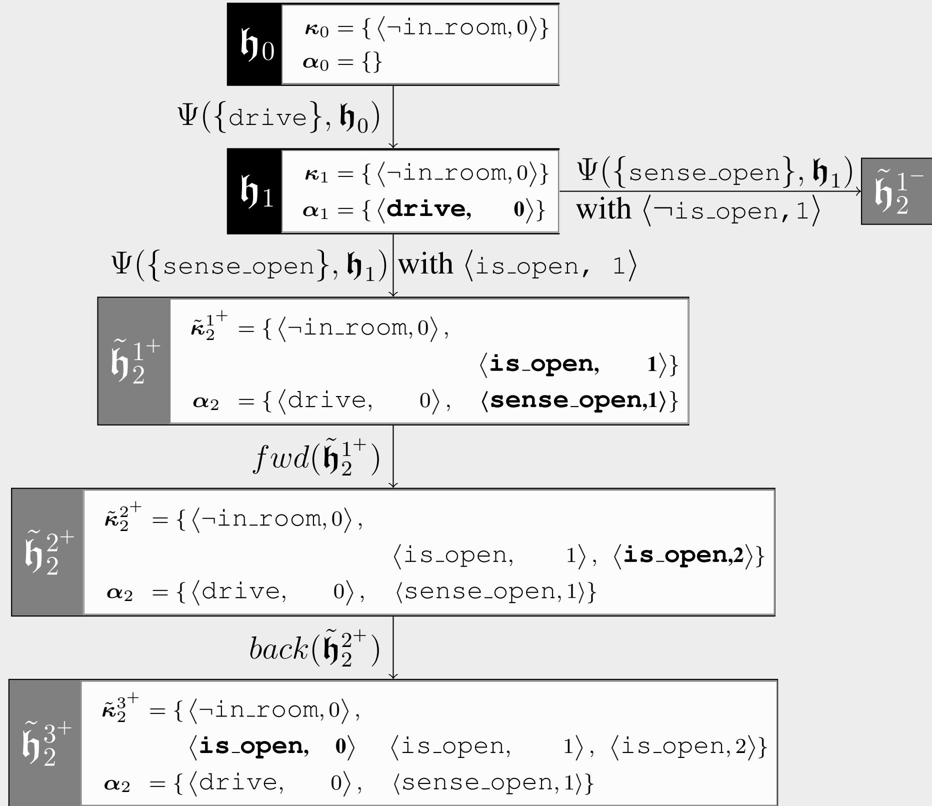
Example 3.2 demonstrates how inertia produces knowledge in the domain specified by Listing 3.2.[2]

```
(:action drive          :effect if is_open then in_room)
(:action sense_open     :observe is_open)
(:init ¬in_room)
(:goal weak in_room)
```

Listing 3.2: Driving through a potentially closed door

[2]Example 3.2 considers that the robot first drives through a door without knowing its open-state and then senses the door's open-state. In practice it makes much more sense to first sense the door's open-state and then drive through the door. We consider this less sensible case any ways to illustrate how forward and backward inertia retrospectively generate knowledge.

Example 3.2 Knowledge gain through sensing and inertia
Consider Listing 3.2 and the action sequence `[drive ; sense_open]`. The state $\mathfrak{h}_0$ is the initial state where the door's open state is uknown. The state transition from $\mathfrak{h}_0$ to $\mathfrak{h}_1$ represents that the robot drives through the door without actually knowing whether it is open.



State $\mathfrak{h}_1$ is similar to $\mathfrak{h}_0$ because the agent does not know whether the door was open at the time of driving. Therefore it does not gain any new knowledge.
`sense_open` generates two intermediate successor h-states: $\tilde{\mathfrak{h}}_2^{1^+}$ contains the positive sensing outcome $\langle \texttt{is_open}, 1\rangle$ and $\tilde{\mathfrak{h}}_2^{1^-}$ contains the negative outcome $\langle \neg\texttt{is_open}, 1\rangle$. For brevity we only consider the positive case.

Forward inertia generates the next intermediate h-state $\tilde{\mathfrak{h}}_2^{2^+}$. Consider that no action is applied that could change `is_open`. Therefore we have that $inertial(\texttt{is_open}, 1, \tilde{\mathfrak{h}}_2^{2^+})$ holds (3.10). Consequently, fwd (3.11) generates knowledge that the door is open in the future and adds $\langle \texttt{is_open}, 2\rangle$ to $\tilde{\boldsymbol{\kappa}}_2^{2^+}$.

The case for next state $\tilde{\mathfrak{h}}_2^{3^+}$ is similar. Since $\neg\texttt{is_open}$ is inertial at step 0 we have that $back$ (3.12) generates $\langle \texttt{is_open}, 0\rangle$.

► IM.3: Causation

We define a function $cause$ that produces knowledge about the effects of actions if the conditions are known. Intuitively, if an effect proposition ep is applied at t all condition literals of ep are known to hold at t then the effect literal l^e of ep holds at step $t+1$.

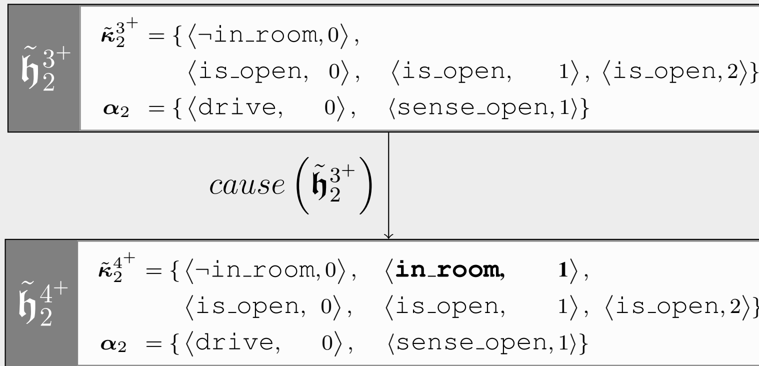
$$cause(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{cause}(\mathfrak{h}) \rangle$$
$$\text{where}$$
$$add_{cause}(\mathfrak{h}) = \{\langle l^e, t\rangle \,|\exists \langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}) : \{\langle l_1^c, t-1\rangle, \ldots, \langle l_n^c, t-1\rangle\} \subseteq \boldsymbol{\kappa}(\mathfrak{h})\} \tag{3.13}$$

with $c(ep) = \{l_1^c, \ldots, l_k^c\}$ and $e(ep) = l^e$.
Example (3.3) illustrates how $cause$ produces knowledge.

Example 3.3 Knowledge gain through causation
Recall the specification of the `drive` action from Listing 3.2:

```
(:action drive :effect if is_open then in_room)
```

Further, reconsider state $\tilde{\mathfrak{h}}_2^{3^+}$ from Example 3.2. We have that $\langle \texttt{open}, 0\rangle$ holds in $\tilde{\mathfrak{h}}_3^{3^+}$ and the action history $\boldsymbol{\alpha}_2$ contains information that `drive` was executed at step 0. Consequently there is an effect proposition $ep_0(\texttt{drive})$ such that $\langle ep_0(\texttt{drive}), 0\rangle \in \boldsymbol{\epsilon}_2$ (see Definition 3.1). The effect proposition has an effect literal $e(ep_0(\texttt{drive})) = \texttt{in_room}$.
$cause$ retrospectively evaluates the effects of the effect proposition. In this case we have $\langle \texttt{in_room}, 1\rangle \in cause\left(\tilde{\mathfrak{h}}_2^{3^+}\right)$.



► IM.4 – IM.5: Positive and negative postdiction

The function pd^{pos} (3.14) defines *positive postdiction*. This is the inference that knowledge about the conditions of an effect proposition is gained if *(a)* the effect is known to

hold after the action and *(b)* known not to hold before the action and *(c)* no other effect proposition could have triggered the effect.

$$\begin{aligned}
&pd^{pos}(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{pd^{pos}}(\mathfrak{h})\rangle \\
&\text{where} \\
&add_{pd^{pos}}(\mathfrak{h}) = \{\langle l^c, t\rangle \,|\exists \langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}) : \\
&\qquad\qquad l^c \in c(ep) \wedge \langle l^e, t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \wedge \langle \overline{l^e}, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \\
&\qquad\qquad \wedge (\forall \langle ep', t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}) : (ep' = ep \vee e(ep') \neq l^e))\} \\
&\text{with } l^e = e(ep)
\end{aligned} \tag{3.14}$$

The function pd^{neg} (3.15) describes *negative postdiction*. This is the inference that knowledge about one yet unknown condition literal of an effect proposition is gained if the effect is known not hold after the action a and all other condition literals are known to hold before the action.

$$\begin{aligned}
&pd^{neg}(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{pd^{neg}}(\mathfrak{h})\rangle \\
&\text{where} \\
&add_{pd^{neg}} = \{\langle \overline{l^c_u}, t\rangle \,|\exists \langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}) : \\
&\qquad\qquad l^c_u \in c(ep) \wedge \langle \overline{l^e}, t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \\
&\qquad\qquad \wedge (\forall l^c \in c(ep) \setminus l^c_u : \langle l^c, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}))\} \\
&\text{with } l^e = e(ep)
\end{aligned} \tag{3.15}$$

Example 3.4 illustrates how knowledge is gained through postdiction.

3.2.8. Re-evaluation of Knowledge-level Effects

To collectively apply the five inference mechanisms in one function we define an $evalOnce$ function that successively applies each of the inference mechanisms.

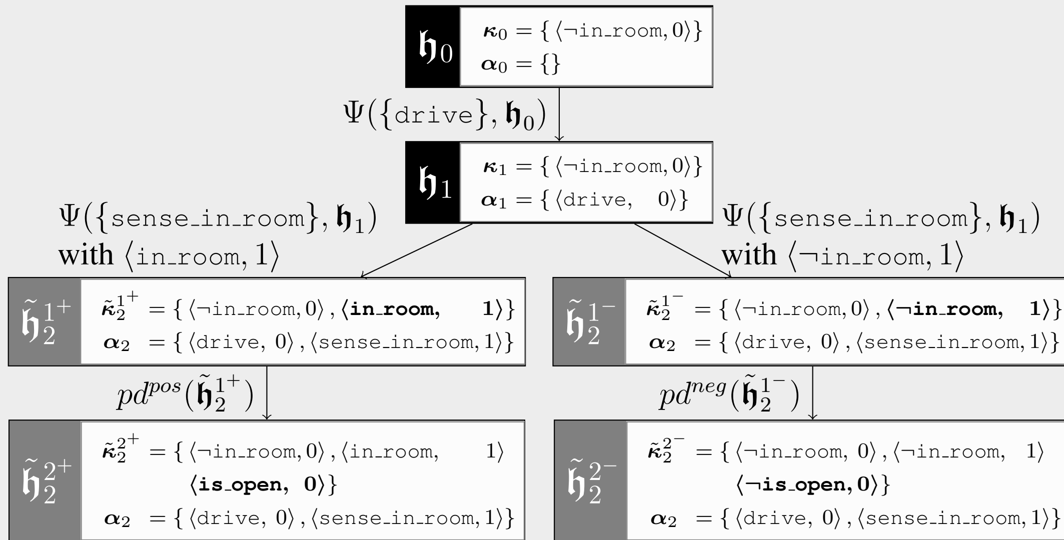
$$evalOnce(\mathfrak{h}) = pd^{neg}(pd^{pos}(cause(back(fwd(\mathfrak{h}))))) \tag{3.16}$$

A problem is that inference mechanism may trigger each other in any order, so it is often not sufficient to apply **IM.1 – IM.5** only once. This is illustrated in Example 3.5. After sensing that the robot arrived in the room postdiction rules are applied to infer that $\langle \texttt{is_open}, 0\rangle \in \tilde{\boldsymbol{\kappa}}_2^{2^+}$. Thereafter inertia rules yield that $\langle \texttt{is_open}, 1\rangle \in \tilde{\boldsymbol{\kappa}}_2^{3^+}$. To this end, re-evaluation is defined recursively (3.17) until convergence is reached.

$$eval(\mathfrak{h}) = \begin{cases} \mathfrak{h} & \text{if } evalOnce(\mathfrak{h}) = \mathfrak{h} \\ eval(evalOnce(\mathfrak{h})) & \text{otherwise} \end{cases} \tag{3.17}$$

Example 3.4 Knowledge gain through postdiction

Consider the domain specified in Listing 3.3 and the sequence `[drive; sense_in_room]`. In the state transition from $\mathfrak{h}_0$ to $\mathfrak{h}_1$ no knowledge is gained because the condition of the `drive` action is not known to hold. For the next state transition we have two branches. According to the transition function (3.7), `sense_in_room` generates two intermediate h-states denoted $\tilde{\mathfrak{h}}_2^{1^+}$ and $\tilde{\mathfrak{h}}_2^{1^-}$. In $\tilde{\mathfrak{h}}_2^{1^+}$ the robot is considered to be in the room, i.e. $\tilde{\mathfrak{h}}_2^{1^+} \models \langle \mathtt{in_room}, 1\rangle$ and in $\tilde{\mathfrak{h}}_2^{1^-}$ it is not in the room, i.e. $\tilde{\mathfrak{h}}_2^{1^-} \models \langle \neg\mathtt{in_room}, 1\rangle$.



In $\tilde{\mathfrak{h}}_2^{2^+}$ it is known that the robot was not in the room when the driving started ($\langle \neg\mathtt{in_room}, 0\rangle \in \tilde{\boldsymbol{\kappa}}_2^{1^+}$) but it was in the room after the driving ($\langle \mathtt{in_room}, 1\rangle \in \tilde{\boldsymbol{\kappa}}_2^{1^+}$). Consequently, positive postdiction generates knowledge that the condition of the `drive`-action was true, i.e. $\langle \mathtt{is_open}, 0\rangle \in add_{pd^{pos}}(\tilde{\mathfrak{h}}_2^{1^+})$.

In $\tilde{\mathfrak{h}}_2^{2^-}$ it is known that the robot was not in the room after the driving (i.e. $\langle \neg\mathtt{in_room}, 1\rangle \in \tilde{\boldsymbol{\kappa}}_2^{1^-}$). Consequently, negative postdiction generates knowledge that the condition of the `drive`-action was false, i.e. $\langle \neg\mathtt{is_open}, 0\rangle \in add_{pd^{neg}}(\tilde{\mathfrak{h}}_2^{1^-})$.

```
(:action drive :effect if is_open then in_room)
(:action sense_in_room :observe in_room)
(:init ¬in_room)
(:goal weak in_room)
```

Listing 3.3: Postdict open-state of door

The recursive definition of $eval$ (3.17) also implies that the order in which the IM are applied in $evalOnce$ (B.2) is arbitrary: as long as all IM are applied in any fixed order, $eval$ will yield the same result.
$eval$ converges in linear time because there exists only a linear number of elements in the knowledge history and because no element is ever removed from the knowledge history (see Lemma B.5).

3.2.9. Concurrent Conditional Plans

So far we considered single state transition steps. In order to model more complex transition models which imply several transition steps we define concurrent conditional plans (CCP). A CCP is a combination of sequences of concurrent actions and `if-then-else` constructs formalized in Definition 3.5.[3]

Definition 3.5 (Concurrent Conditional Plan)

- An empty sequence of actions (denoted by `[]`) is a CCP
- If $a_1, \ldots, a_n$ are actions, then $[a_1||\ldots||a_n]$ is a CCP.
- If p_1 and p_2 are concurrent conditional plans, then $[p_1;p_2]$ is a CCP.
- If p_1 and p_2 are concurrent conditional plans and l is a fluent literal, then `[if` l `then` p_1 `else` p_2`]` is a CCP[4]

Listing 3.4 specifies a planning domain together with a plan p_1 which solves the problem. The weak goal[5] is to get into an adjacent room and it is uncertain whether opening the door to the room will succeed: the agent first tries to open a door, then verifies whether the door is indeed open and drives through the door only if opening succeeded.

```
(:action drive       :effect if is_open then in_room)
(:action open_door   :effect if ¬jammed then is_open)
(:action sense_open  :observe is_open)
(:init ¬in_room)
(:goal weak in_room)

p1 = [open_door; sense_open; [if open then [drive]]]
```

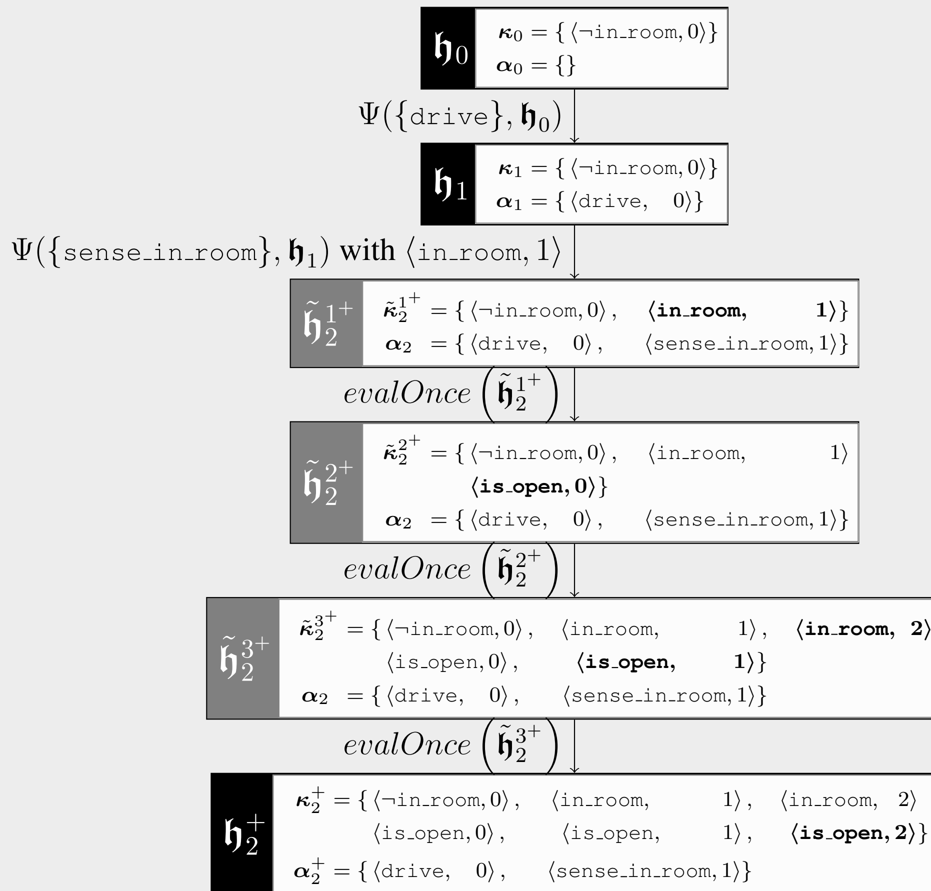
Listing 3.4: A problem which requires postdiction with conditional plan

[3]Recall that we use a restricted form of concurrency where all actions have the same duration. Hence we consider only concurrent actions and not concurrent (sub-)plans.

[4]For notational convenience we allow to skip the `else`-part, i.e. `[if` l `then` p_1`]` is equivalent to `[if` l `then` p_1 `else []]`

[5]Recall that weak goals must only be achieved in at least one leaf of the transition tree. See Section 2.1.1 for details.

Example 3.5 Repeated Evaluation
Reconsider Listing 3.3 and the sequence `[drive; sense_in_room]`. State $\mathfrak{h}_1$ which results from the `drive` action does not contain additional knowledge because the condition of the drive action (the door being open) is unknown.
Sensing generates an intermediate successor state $\tilde{\mathfrak{h}}_2^{1^+}$ state with $\tilde{\mathfrak{h}}_2^{1^+} \models \langle \texttt{in_room}, 1 \rangle$. Thereafter, $evalOnce(\tilde{\mathfrak{h}}_2^{1^+})$ (B.2) calls **IM.1** – **IM.5**. The only IM that produces knowledge in this first evaluation step is *positive postdiction* (3.14) – **IM.4** which adds a pair $\langle \texttt{is_open}, 0 \rangle$. This results in the next intermediate h-state $\tilde{\mathfrak{h}}_2^{2^+}$.
In the next re-evaluation step, $evalOnce(\tilde{\mathfrak{h}}_2^{2^+})$ calls **IM.1** – **IM.5** again, and *forward inertia* (3.11) – **IM.1** generates knowledge that the door is open during the sensing: $\langle \texttt{is_open}, 1 \rangle$. It also generates knowledge that the robot is in the room after the sensing: $\langle \texttt{in_room}, 2 \rangle$. A third application of $evalOnce$ results in the state $\mathfrak{h}_2^+$. Here, forward inertia generates knowledge that the door is open after the sensing: $\langle \texttt{is_open}, 2 \rangle$. The state $\mathfrak{h}_2^+$ is not an intermediate state because further application of $evalOnce$ will not produce any additional knowledge, i.e. the re-evaluation process converged.



3.2.10. Extended Transition Function

We define an extended transition function $\widehat{\Psi}$ that maps a plan and a state to a set of states.

$$\widehat{\Psi}(p, \mathfrak{h}) = \begin{cases} \{\mathfrak{h}\} & \text{if } p = \texttt{[]} \\ \Psi(\{a_1 \dots a_n\}, \mathfrak{h}) & \text{if } p = \texttt{[}a_1 \texttt{||} \dots \texttt{||} a_n\texttt{]} \\ \bigcup\limits_{\mathfrak{h}' \in \widehat{\Psi}(p_1, \mathfrak{h})} \widehat{\Psi}(p_2, \mathfrak{h}') & \text{if } p = \texttt{[}p_1\texttt{;}p_2\texttt{]} \\ \widehat{\Psi}(p_1, \mathfrak{h}) & \text{if } p = \texttt{if } l \texttt{ then } p_1 \texttt{ else } p_2 \text{ and } \mathfrak{h} \models l \\ \widehat{\Psi}(p_2, \mathfrak{h}) & \text{if } p = \texttt{if } l \texttt{ then } p_1 \texttt{ else } p_2 \text{ and } \mathfrak{h} \not\models l \end{cases} \tag{3.18}$$

The extended transition function models *branching* as a reaction on respective sensing results. For instance, consider an initial state $\mathfrak{h}_0$ and plan p_1 from Listing 3.4. Given that $\mathfrak{h}_0$ does not contain information about the open-state of the door (e.g. $\boldsymbol{\kappa}(\mathfrak{h}_0) = \emptyset$), `sense_open` will generate two h-states: one where the resulting h-state satisfies the condition `is_open` and another where it does not satisfy the condition. In the former h-state – where the condition is satisfied – the action `drive` is applied after sensing. In the latter h-state action execution ends after sensing. Hence, the extended transition function for p_1 in Listing 3.4 evaluates as follows:

$$\begin{aligned} \widehat{\Psi}(p_1, \mathfrak{h}_0) = \{&\Psi(\texttt{drive}, \Psi(\texttt{sense_open}, \Psi(\texttt{open_door}, \mathfrak{h}_0))), \\ &\Psi(\texttt{sense_open}, \Psi(\texttt{open_door}, \mathfrak{h}_0))\} \end{aligned} \tag{3.19}$$

A detailed trace of how (3.19) generates the transition tree can be found in Appendix D.2.

3.2.11. Plan Verification – Weak and Strong Goals

The extended transition function takes a plan p and an initial h-state $\mathfrak{h}_0$ of a domain $\mathcal{D}$ as input and generates a transition tree. The nodes of the three are h-states and its edges are actions and sensing results.

Weak goals require that a goal is *possibly* achieved by a plan. That is, it is sufficient to have one leaf of the transition tree where the goal literal is known to hold. In contrast, *strong goals* require that a literal is known to hold in *all* leafs of the transition tree. This is implemented by the plan verification function (3.20).

$$\begin{aligned} solves(p, \mathcal{D}) = \quad &\forall \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : \forall l^{sg} \in \mathcal{G}^{strong} : \mathfrak{h} \models l^{sg} \\ &\wedge \exists \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : \forall l^{wg} \in \mathcal{G}^{weak} : \mathfrak{h} \models l^{wg} \end{aligned} \tag{3.20}$$

where $\mathcal{G}^{strong}$, $\mathcal{G}^{weak}$ are the goal proposition in the planning domain $\mathcal{D}$. Consider a concurrent conditional plan p and an initial h-state $\mathfrak{h}_0$. The leaf states of the transition tree are generated by calling the extended transition function $\widehat{\Psi}(p, \mathfrak{h}_0)$. (3.20) involves a

$\forall$-quantification to state that strong goals (denoted by l^{sg}) must hold in all leafs of the transition tree. Weak goals (denoted by l^{wg}) must only hold in one leaf (expressed with the $\exists$-quantifier).

3.3. Computational Complexity of $\mathcal{HPX}$

For determining the computational complexity we only consider plans of polynomial size wrt. the input problem, i.e. the size of p is polynomial wrt. the size of $\mathcal{D}$. This restriction is justified because plans which grow exponentially wrt. to the planning problem are not useful in practice (a similar argument can be found in (Baral et al., 2000)). The following theorem states that under this restriction solving the plan existence problem is in NP:

Theorem 3.1 (Complexity of the $\mathcal{HPX}$ planning problem) *Given a planning domain $\mathcal{D}$, deciding whether the following holds is in NP:*

$$\exists p : solves(p, \mathcal{D}) \tag{3.21}$$

Proof sketch: (The full proof can be found in Appendix B.1)

- Let p be a plan of which the size is polynomial wrt. the size of a domain $\mathcal{D}$. Then $solves(p, \mathcal{D})$ is polynomial for to the following reasons:
  - $solves(p, \mathcal{D})$ calls the extended transition function $\widehat{\Psi}(p, \mathfrak{h}_0)$. For each set of actions $\boldsymbol{A}$ in p the transition function $\Psi(\boldsymbol{A}, \mathfrak{h})$ is called for an h-state $\mathfrak{h}$. Since p is of polynomial size, this happens polynomially often (see Lemma B.1).
  - Calling $\Psi(\boldsymbol{A}, \mathfrak{h})$ involves calling the re-evaluation function $eval$ (3.17), which in turn calls $evalOnce(\mathfrak{h})$ (B.2) until the h-state converged. $evalOnce(\mathfrak{h})$ converges after a polynomial number of applications because the size of the knowledge history $\boldsymbol{\kappa}(\mathfrak{h})$ is linear wrt. the length of the plan p and we restrict p to be of polynomial size.
  - A single re-evaluation step $evalOnce(\mathfrak{h})$ employs inference mechanisms **IM.1–IM.5**. These are all performed in polynomial time (Lemma B.4).
- If $solves(p, \mathcal{D})$ is polynomial for a given p then deciding whether $\exists p : solves(p, \mathcal{D})$ is in NP. ■

3.4. Relation Between $\mathcal{HPX}$ and $\mathcal{PWS}$: A Temporal Semantics for the Action Language $\mathcal{A}_k$

In order to describe how $\mathcal{HPX}$ relates to traditional epistemic action theories which are based on a possible-worlds-semancics ($\mathcal{PWS}$) we present a *temporal query semantics*

for the action language $\mathcal{A}_k$ (Son and Baral, 2001) which we call $\mathcal{A}_k^{TQS}$.
$\mathcal{A}_k^{TQS}$ is a $\mathcal{PWS}$-based approach to reason about the past and can express statements like "at step 5 it is known that the door was open at step 3". This is also possible with $\mathcal{HPX}$, but not with traditional $\mathcal{PWS}$-based semantics like the original $\mathcal{A}_k$. Theorem 3.3 states that $\mathcal{HPX}$ is sound wrt. $\mathcal{A}_k^{TQS}$.
Note that we want to keep the focus on the temporal postdiction aspect of knowledge and therefore we make the following simplifications: *(a)* we do not consider executability conditions and initial state constraints, *(b)* we restrict the $\mathcal{A}_k$ semantics and forbid sensing the value of more than one single fluent per action and *(c)* we only consider sequences of actions. We justify these simplifications with the argument that they do not affect the temporal postdiction mechanics which we are interested in.

3.4.1. Syntactical Mapping Between $\mathcal{A}_k$ and $\mathcal{HPX}$

The syntactical mapping between the original action language $\mathcal{A}_k$ (Son and Baral, 2001) and $\mathcal{HPX}$ is presented in Table 3.1. This illustrates that an $\mathcal{A}_k$ domain description D can always be mapped to a corresponding $\mathcal{HPX}$ domain specification due to the one-to-one syntactical correspondence.

	$\mathcal{A}_k$	$\mathcal{HPX}$ PDDL dialect
Value prop.	**initially**(l^{init})	`(:init` l^{init}`)`
Effect prop.	**causes**(a,l^e, $\{l_1^c \dots l_n^c\}$)	`(:action` a `:effect if (and` $l_1^c \dots l_k^c$`) then` l^e`)`
Sensing	**determines** (a,$\{f^s$,$\neg f^s\}$)	`(:action` a `:observe` f^s`)`

Table 3.1.: Relation between $\mathcal{A}_k$ syntax and our PDDL dialect

The set of effect propositions (EP) of an action a (denoted $\mathcal{EP}^a$) and the knowledge proposition (KP) of an action a (denoted $\mathcal{KP}^a$) is obtained analogously to the case of the PDDL-syntax. For example **causes**(a,l^e, $\{l_1^c \dots l_k^c\}$) is semantically represented as an effect proposition ep with the effect literal $e(ep) = f^e$ and the condition literals $c(ep) = \{l_1^c, \dots, l_k^c\}$. Similarly, **determines** (a,$\{f^s$,$\neg f^s\}$) denotes that action a has the knowledge proposition f^s, denoted $\mathcal{KP}^a = f$.

3.4.2. Original $\mathcal{PWS}$-based $\mathcal{A}_k$ Semantics

The original $\mathcal{A}_k$ semantics (Son and Baral, 2001) is defined via a transition function that maps actions and so-called *c-states* to c-states. A c-state δ is a tuple $\langle u, \Sigma \rangle$, where u is called a *state* and Σ is called a *k-state*. Informally, u represents a possible world and Σ represents the possible agent's belief wrt. u. A k-state Σ is a set of possible states, denoted $s \in \Sigma$. A state (denoted u or s) is a set of fluents. If for a state s and a fluent f it

holds that $f \in s$ then the value of f in s is $true$ and otherwise $false$. We require that c-states are *grounded*, i.e. that $u \in \Sigma$ for all c-states $\delta = \langle u, \Sigma \rangle$. Intuitively this means that the possible world u is among the worlds Σ the agent believes it could be in. For convenience we introduce the following $\models$-notation for a state s:

$$\begin{aligned} (s \models f) &\Leftrightarrow (f \in s) \\ (s \models \neg f) &\Leftrightarrow (f \notin s) \\ (s \models \mathcal{L} &\Leftrightarrow (\forall l \in \mathcal{L} : s \models l) \end{aligned} \tag{3.22}$$

where $\mathcal{L}$ is a set of literals. Similarly we define for a k-state Σ:

$$(\Sigma \models l) \Leftrightarrow (\forall s \in \Sigma : s \models l) \tag{3.23}$$

Equation (3.23) reflects that a literal l is known to hold if it is $true$ in all possible worlds s in Σ. Given a domain description $\mathcal{D}$, one is interested in a set $\mathcal{M}$ of *valid models* of $\mathcal{D}$. A valid model $m = \langle \delta_0, \Phi \rangle$ is a pair of a *valid initial c-state* δ_0 and a transition function Ψ. An initial c-state is called *valid* if it does not contradict the initial knowledge defined in $\mathcal{D}$ (see (Son and Baral, 2001) for details).
The transition function emerges from the effect propositions and knowledge propositions in $\mathcal{D}$. It maps an action a and a c-state to a c-state. It is defined with a case distinction as follows:

1. ***a* is a non-sensing action:** in this case the transition function is defined as:

$$\Phi(a, \langle u, \Sigma \rangle) = \big\langle Res(a, u), \{Res(a, s') | s' \in \Sigma\} \big\rangle \text{ where} \tag{3.24}$$

$$Res(a, s) = s \cup E_a^+(s) \setminus E_a^-(s) \text{ where} \tag{3.25}$$

$$\begin{aligned} E_a^+(s) &= \{f \mid \exists ep \in \mathcal{EP}^a : e(ep) = f \wedge s \models c(ep)\} \\ E_a^-(s) &= \{\neg f \mid \exists ep \in \mathcal{EP}^a : e(ep) = f \wedge s \models c(ep)\} \end{aligned}$$

   Res (3.25) is a result function that reflects causation: if all conditions of an effect proposition ep hold (denoted $s \models c(ep)$), then the effect $e(ep)$ holds in the result.

2. ***a* is a sensing action:** in this case a has a knowledge proposition $\mathcal{KP}^a = f^s$ and the transition function is defined as:

$$\Phi(a, \langle u, \Sigma \rangle) = \langle u, \{s | (s \in \Sigma) \wedge (f^s \in s \Leftrightarrow f^s \in u)\} \rangle \tag{3.26}$$

   Intuitively, (3.26) rules out these possible worlds in Σ which do not coincide with the actual world u.

Example 3.6 illustrates the original $\mathcal{A}_k$ semantics.

Example 3.6 $\mathcal{PWS}$-based $\mathcal{A}_k$ semantics
Consider Figure 3.1. A robot can execute an action `drive` to get into the living room if the door to the room is open. A fluent `in_liv` denotes that it is in the living room and a fluent `is_open` denotes that the door is open. A sensing action `sense_in_liv` can be executed to determine whether or not the robot is in the living room. The domain specification in $\mathcal{A}_k$ syntax is:

$$\textbf{initially}(\neg \texttt{in_liv})$$
$$\textbf{causes}(\texttt{drive}, \texttt{in_liv}, \{\texttt{is_open}\})$$
$$\textbf{determines}(\texttt{sense_in_liv}, \{\texttt{in_liv}, \neg\texttt{in_liv}\})$$

Initially it is known that the robot is not in the living room and it is unknown whether the door is open. This results in two valid initial c-states: $\delta_{0:a} = \langle u_{0:a}, \Sigma_{0:a}\rangle$ and $\delta_{0:b} = \langle u_{0:b}, \Sigma_{0:b}\rangle$. $u_{0:a}$ and $u_{0:b}$ represent two initial possible worlds where the door may be open or not, i.e. $u_{0:a} = \{\texttt{open}\}$ and $u_{0:b} = \{\}$. $\Sigma_{0:a}$ and $\Sigma_{0:b}$ represent two possible knowledge-states wrt. the possible worlds $u_{0:a}$ and $u_{0:b}$. Initially these are identical, i.e. $\Sigma_{0:a} = \Sigma_{0:b}$.
Applying the transition function (3.24) with `drive` results in the next states $\delta_{1:a}$ and $\delta_{1:b}$. Here we have that $u_{1:a} = \{\texttt{open}, \texttt{in_liv}\}$ and $u_{1:b} = \{\}$. In the second possible world $u_{1:b}$ the robot is stuck in front of a closed door. The knowledge-states in the two possible worlds are again identical because so far no sensing has happened, i.e. $\Sigma_{1:a} = \Sigma_{1:b}$.
To model the sensing action the transition function (3.26) generates the next c-states $\delta_{2:a}$ and $\delta_{2:b}$. The sensing "transfers" information about being in the living room or not from the actual world to the knowledge state. This is done by eliminating these states which are "incompatible" with the possible worlds $u_{1:a}$, resp. $u_{1:b}$. This causes two possible knowledge states wrt. each individual possible world, $\Sigma_{2:a} = \{\{\texttt{open}, \{\texttt{in_liv}\}\}$ and $\Sigma_{2:b} = \{\}\}$. In $\Sigma_{2:a}$ state it is known that the robot is in the living room and the door is open and in $\Sigma_{2:b}$ it is known that the robot is stuck at the closed door, not being in the living room.

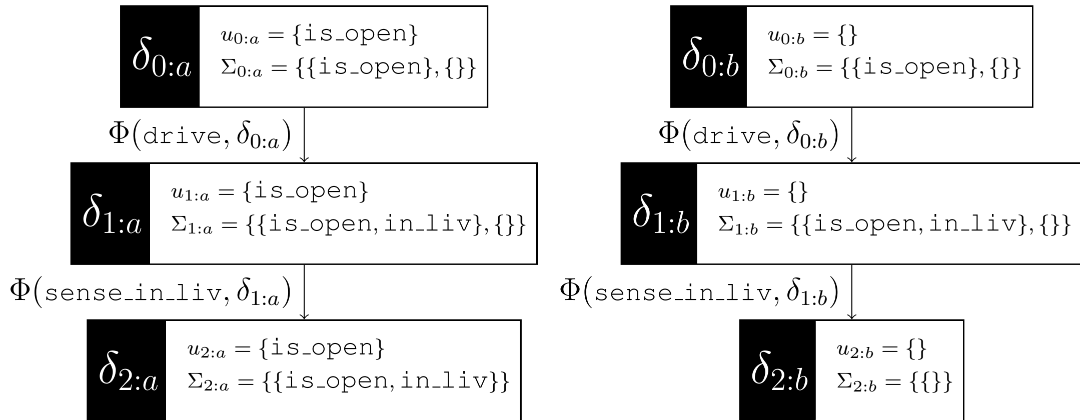


Figure 3.1.: $\mathcal{A}_k$ semantics

3.4.3. Temporal Query Semantics – $\mathcal{A}_k^{TQS}$

Our approach to make $\mathcal{A}_k$ capable of temporal reasoning is based on a re-evaluation step with an intuition is as follows: Let $\Sigma_0 = \{s_0^0, \ldots, s_0^{|\Sigma_0|}\}$ be the set of all possible initial states of a (complete) initial k-state wrt. a valid initial c-state δ_0 of an $\mathcal{A}_k$ domain $\mathcal{D}$. Whenever sensing happens, the transition function will remove some states from the k-state, i.e. $\Phi(a_n, \Phi(a_{n-1}, \cdots \Phi(a_1, \langle u_0, \Sigma_0 \rangle))) = \langle u_n, \Sigma_n \rangle$. To reason about the past, we *refine the set of possible initial states* and re-apply the result function to the refined set of initial states again. The refined set of initial states is the set of initial states which "survived" the transition of a sequence of actions.
For example, consider a sequence of n actions and say we are interested in the world state after the t-th action. Then we consider a *re-evaluated initial k-state*, denoted Σ_0^n, which consists of states $s_0 \in \Sigma_0$ such that for $s_n = Res(a_n, \cdots Res(a_1, s_0)$ it holds that $s_n \in \Sigma_n$. In other words, we consider these initial states s_0 of which the child states s_n "survived" the sensing actions.
Once we identified the re-evaluated initial k-state we apply the result function on each $s \in \Sigma_0^n$ up to the t-th action for this state. The resulting *re-evaluated k-state* is denoted Σ_n^t. If a fluent holds in all states $s_n^t \in \Sigma_n^t$, then after the n-th action, it is known that a fluent holds after the t-th action. This is formalized by Definitions 3.6 and 3.7.

Definition 3.6 (Re-evaluated initial k-state) *Let $\alpha = [a_1; \ldots; a_n]$ be a sequence of actions and $\delta_0 = \langle u_0, \Sigma_0 \rangle$ be a valid initial c-state such that $\langle u_n, \Sigma_n \rangle = \Phi(a_n, \Phi(a_{n-1}, \cdots \Phi(a_1, \delta_0)))$. We define a re-evaluated initial k-state, denoted Σ_n^0, as the set of initial belief states in Σ_0 which are valid after applying α:*[6]

$$\Sigma_n^0 = \{s_0 | s_0 \in \Sigma_0 \wedge Res(a_n, Res(a_{n-1}, \cdots Res(a_1, s_0))) \in \Sigma_n\} \tag{3.27}$$

Re-evaluated c-states are defined in Definition 3.7: given a sequence of actions α, re-evaluated c-states are obtained by applying the $\mathcal{A}_k$ transition functions (3.24) and (3.26) on the re-evaluated initial k-state Σ_0^n.

Definition 3.7 (Re-evaluated c-states) *Let $\alpha = [a_1; \ldots; a_n]$ be a sequence of actions and $\delta_0 = \langle u_0, \Sigma_0 \rangle$ be a valid initial c-state, such that Σ_n^0 is a re-evaluated initial k-state according to Definition 3.6. We define a re-evaluated c-state, denoted δ_n^t as follows*

$$\delta_n^t = \left\langle u_n, \Sigma_n^t \right\rangle \tag{3.28}$$

$$\textit{where} \quad u_n = Res(a_n, u_{n-1}) \quad \textit{and} \quad \Sigma_n^t = \bigcup_{s \in \Sigma_n^{t-1}} Res(a_t, s) \tag{3.29}$$

with $0 < t \leq n$

Σ_n^t is called the re-evaluated k-state.

[6]Consider that according to (3.25) $Res(a, s) = s$ if a is a sensing action.

A property that emerges from Definition 3.7 is that knowledge itself is persistent: if after n actions it is known that l holds at t, then this is still known after $n+1$ actions (with $0 \leq t \leq n$). That is, $\Sigma^t_{n+1} \subseteq \Sigma^t_n$ (see Lemma C.5 in Appendix C for a formal proof).

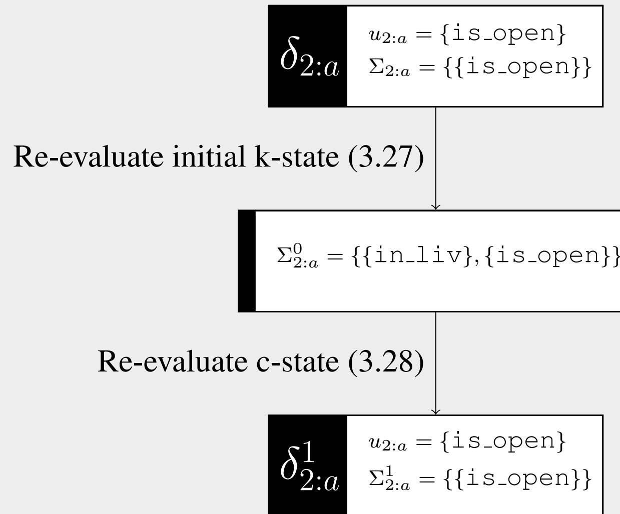
Example 3.7 $\mathcal{A}_k{}^{TQS}$ **semantics**
Consider the c-state $\delta_{2:a}$ from Example 3.6. The original $\mathcal{A}_k$ semantics allows one to infer that after the sensing the robot knows that the door is open and that it is in the living room, i.e. $\Sigma_{2:a} = \{\texttt{is_open}, \texttt{in_liv}\}$. However, it does not allow one to infer whether the robot knows that the door was already open before the sensing, i.e. at a step $t = 1$. Our Temporal Query Semantics supports this inference as follows: Let $\delta_{2:a} = \langle u_{2:a}, \Sigma_{2:a}\rangle = \langle\{\texttt{is_open}, \texttt{in_liv}\}, \{\{\texttt{is_open}, \texttt{in_liv}\}\}\rangle$. Then applying (3.27) for re-evaluated initial k-states yields

$$\Sigma^0_{2:a} = \{s_0 \in \Sigma_{0:a} | Res(\texttt{sense_in_liv}, Res(\texttt{drive}, s_0)) \in \Sigma_{2:a}\} = \{\texttt{is_open}\}$$

The re-evaluation step (3.28) produces

$$\Sigma^1_{2:a} = Res(\texttt{drive}, \{\texttt{is_open}\}) = \{\{\texttt{is_open}, \texttt{in_liv}\}\}$$

In other words, $\mathcal{A}_k^{TQS}$ can express the temporal statement “after the sensing the robot knows that it was in the living room before the sensing”.



3.4.4. Relation between $\mathcal{A}_k$ and $\mathcal{A}_k^{TQS}$

Since $\mathcal{A}_k$ does not have a temporal knowledge dimension we can only consider knowledge about the present state to formally relate $\mathcal{A}_k$ to $\mathcal{A}_k^{TQS}$. The following Theorem 3.2 considers equivalence of $\mathcal{A}_k$and ${\mathcal{A}_k}^{TQS}$ for the projection problem for a sequence of actions for the present state, i.e. the case where $t = n$.

Theorem 3.2 (Equivalence of $\mathcal{A}_k$ and $\mathcal{A}_k^{TQS}$ for $t = n$) *Given a domain $\mathcal{D}$, a valid initial c-state $\delta_0 = \langle u_0, \Sigma_0 \rangle$ and a sequence of actions $\alpha_n = [a_1; \ldots ; a_n]$ such that $\langle u_n, \Sigma_n \rangle = \Phi(a_n, \Phi(a_{n-1}, \cdots \Phi(a_1, \langle u_0, \Sigma_0 \rangle)))$. Let $\Sigma_n^n = Res(a_n, Res(a_{n-1}, \cdots Res(a_1, \Sigma_n^0)))$ be a re-evaluated k-state with Σ_n^0 as the re-evaluated initial state according to Definition 3.6. Then (3.30) holds.*

$$\Sigma_n = \Sigma_n^n \tag{3.30}$$

Proof:
The theorem follows directly from Lemma C.6.

3.4.5. Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$

Now that we have defined $\mathcal{A}_k^{TQS}$– a semantics which is *(a)* based on the possible-worlds semantics and *(b)* can express temporal knowledge we can formally relate $\mathcal{HPX}$ with a possible-worlds approach. The following Theorem 3.3 considers soundness of $\mathcal{HPX}$ wrt. ${\mathcal{A}_k}^{TQS}$ for the projection problem for a sequence of actions.
Soundness is defined wrt. an initial h-state $\mathfrak{h}_0$ and an arbitrary valid initial c-state of a domain. On the $\mathcal{A}_k^{TQS}$-side the theorem considers one valid initial c-state u_0, i.e. one possible world which does not contradict the initial knowledge definitions. On the $\mathcal{HPX}$-side we argue that there exists on h-state $\mathfrak{h}_n$ such that if a pair $\langle l, t \rangle$ is known to hold in $\mathfrak{h}_n$ then l is known to hold in the re-evaluated k-state Σ_n^t which results from the valid initial c-state u_0. A similar notion of soundness is presented for (Son and Baral, 2001, Proposition 6, Lemma C.4).

Theorem 3.3 (Soundness of $\mathcal{HPX}$ wrt. ${\mathcal{A}_k}^{TQS}$) *Let $\alpha = [a_1; \ldots ; a_n]$ be a sequence of actions and D a domain specification. Let $\langle u_0, \Sigma_0 \rangle$ be a valid grounded initial c-state of $\mathcal{D}$, such that with Definition 3.7 the re-evaluated c-state after $t \leq n$ actions is given as $\langle u_t, \Sigma_n^t \rangle = \Phi(a_t, \Phi(a_{t-1}, \cdots \Phi(a_1, \langle u_0, \Sigma_n^0 \rangle)))$. Then there exists a h-state $\mathfrak{h}_n \in \widehat{\Psi}(\alpha, \mathfrak{h}_0)$ such that for all literals l and all steps n, t with $0 \leq t \leq n$:*

$$(\mathfrak{h}_n \models \langle l, t \rangle) \Rightarrow (\Sigma_n^t \models l) \tag{3.31}$$

Proof:
The theorem follows directly from Lemma C.1. We proof Lemma C.1 by induction over the number of actions (see Appendix C).

4 Answer Set Programming Formalization of $\mathcal{HPX}$

To make $\mathcal{HPX}$ applicable in practice, we implement the theory in terms of Answer Set Programming. To this end, the individual inference mechanisms **IM.1 – IM.5**, which we present in Chapter 3, are modeled as Logic Programming rules. An additional set of rules is used to implement plan generation and verification.
Section 4.1 describes the main predicates and how the temporal dimension of knowledge is represented. Section 4.2 provides an overview of the constitution of an $\mathcal{HPX}$-Logic Program. This consists of a domain-specific and a domain-independent part. The domain-specific part is generated by eight *translation rules* (T1) – (T8), which compile the PDDL-like input language into LP rules (Section 4.3). These include the inference mechanisms of postdiction and causation. The LP rules which represent the domain-independent part are fixed and model inertia and sensing (Section 4.4). Section 4.5 describes the way in which agents interpret Stable Models of the Logic Program as conditional plans.
The Logic Programming implementation of $\mathcal{HPX}$ constitutes an alternative model-theoretic semantics of $\mathcal{HPX}$. The relation between the model-theoretic semantics and the operational $\mathcal{HPX}$ semantics is illustrated in Section 4.6, which also contains a corresponding soundness theorem.

4.1. Main Predicates and Notation

The following are the main predicates used in the ASP formalization:

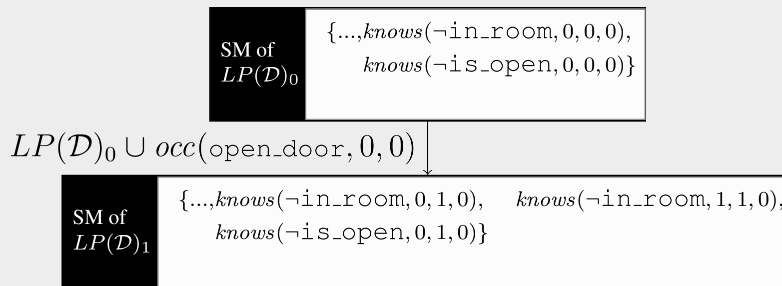
- $knows(l, t, n, b)$ states that at step n in branch b it is known that l holds (or did hold) at step t (with $t \leq n$).

- $occ(a, n, b)$ denotes that action a occurs at step n in branch b.

- $apply(ep, n, b)$ denotes that an effect proposition ep is applied at step n in branch b. Whenever $occ(a, n, b)$ and ep is an effect proposition of a, then $apply(ep, n, b)$. This reflects the abstraction from action histories to effect histories (see Definition 3.1).
- $sRes(l, n, b, b')$ denotes that the literal l is sensed at step n in branch b, such that it will hold in the child branch b'.
- $uBr(n, b)$ denotes that branch b is a valid branch at step n. Actions can only be executed if a branch is valid.

As a notational convention, for negative literals we write $\neg f$ to denote $neg(f)$ in ASP syntax. A *state transition* in terms of the ASP formalization of $\mathcal{HPX}$ can be understood by adding an $occ/3$ atom to the Logic Program. This is illustrated in Example 4.1.

Example 4.1 Action and knowledge history in ASP formalization
Let $\mathcal{D}$ be the domain specified by Listing 4.1. A robot can execute an `open_door` action under consideration that the action may fail (door may be jammed) and the door is in fact not open after execution. The value proposition in Listing 4.1 translates to $knows(\neg\texttt{is_open}, 0, 0, 0)$, i.e. this atom is contained in the Stable Model of an initial Logic Program $LP(\mathcal{D})_0$.
The occurrence of action `open_door` is represented by adding the atom $occ(\texttt{open_door}, 0, 0)$ to the Logic Program $LP(\mathcal{D})_0$, resulting in $LP(\mathcal{D})_1 = LP(\mathcal{D})_0 \cup occ(\texttt{open_door}, 0, 0)$



The reasoning mechanisms must be defined such that the Stable Model of $LP(\mathcal{D})_1$ does not contain knowledge about the door-state after executing `open_door`, because it is unknown whether the door is jammed. Further, the reasoning mechanisms must cover inertia: no action occurred that could have affected the robot's location outside the living room ($\neg$`in_room`) and therefore one can conclude that the robot remains outside the room after opening the door, i.e. $knows(\neg\texttt{in_room}, 1, 1, 0)$.

```
(:action open_door :effect if ¬jammed then is_open)
(:init  ¬is_open)
```

Listing 4.1: Opening a potentially jammed door

A *conditional plan* is determined by a set of occ, and $sRes$ atoms. For example, consider the atoms $occ(a_0, 0, b)$, $sRes(f, 0, b, b)$, $sRes(\neg f, 0, b, b')$, $occ(a_1, 1, b)$ and $occ(a_2, 1, b')$. This is equivalent to the conditional plan a_0`;[if` f `then` a_1 `else` a_2`]`. A detailed formal description of plan extraction from Stable Models is provided in Section 4.5.

4.2. Constitution of an $\mathcal{HPX}$-Logic Program

The formalization is based on a domain independent foundational theory Γ_{hpx} and on a set of *translation rules* **T** that are applied to a planning domain $\mathcal{D}$. An ASP formalization of $\mathcal{D}$, denoted by LP($\mathcal{D}$), consists of a domain dependent theory and a domain independent theory:

- Domain dependent theory (Γ_{world}): It consists of a set of rules Γ_{init} representing initial knowledge; Γ_{act} representing actions; and Γ_{goal} representing goals.
- Domain independent theory (Γ_{hpx}): This consists of a set of auxiliary definitions Γ_{aux}; a set of rules to handle inertia (Γ_{in}); sensing (Γ_{sen}); inference mechanisms (Γ_{infer}); concurrency (Γ_{conc}); plan verification (Γ_{verify}) and plan-generation & optimization (Γ_{plan}).

The resulting Logic Program LP($\mathcal{D}$) is given as:

$$LP(\mathcal{D}) = \underbrace{[\,\Gamma_{aux} \cup \Gamma_{in} \cup \Gamma_{sen} \cup \Gamma_{infer} \cup \Gamma_{conc} \cup \Gamma_{verify} \cup \Gamma_{plan}\,]}_{\Gamma_{hpx}} \cup \underbrace{[\,\Gamma_{init} \cup \Gamma_{act} \cup \Gamma_{goal}\,]}_{\Gamma_{world}} \tag{4.1}$$

We call a Logic Program that is assembled according to (4.1) an $\mathcal{HPX}$-Logic Program for a planning problem $\mathcal{D}$.

4.3. Translation Rules: ($\mathcal{D} \overset{\textbf{T1–T8}}{\longmapsto} \Gamma_{world}$)

The domain dependent theory Γ_{world} is obtained by applying the set of translation rules $\mathbf{T} = \{T1, \ldots, T8\}$ on a planning domain $\mathcal{D}$, specified in our PDDL-like input syntax. Recall the following syntactical elements from Chapter 3:

$$\texttt{(:init (and } l_1^{init} \texttt{...} l_{n_{in}}^{init}\texttt{))} \quad (3.1a)$$
$$\texttt{(oneof } l_1^{isc} \texttt{...} l_n^{isc}\texttt{)} \quad (3.1b)$$
$$\texttt{(:action } a \texttt{ :effect if (and } l_1^c \texttt{...} l_{n_c}^c\texttt{) then } l^e\texttt{)} \quad (3.1c)$$
$$\texttt{(:action } a \texttt{ :observe } f\texttt{)} \quad (3.1d)$$
$$\texttt{(:action } a \texttt{ :executable (and } l_1^{ex} \texttt{...} l_{n_{ex}}^{ex}\texttt{))} \quad (3.1e)$$
$$\texttt{(:goal } \text{type} \texttt{ (and } l_1^g \texttt{...} l_{n_g}^g\texttt{))} \quad (3.1f)$$

The translation rules are as follows:

Action and Fluent Declarations (T1)

For every fluent f or action a, LP($\mathcal{D}$) contains the facts:

$$\mathit{fluent}(f). \quad \mathit{action}(a). \tag{T1}$$

Initial Knowledge ($\mathcal{I} \overset{\text{(T2)–(T3}}{\longmapsto} \Gamma_{init}$)

Facts Γ_{init} for initial knowledge are obtained by applying translation rule (T2). For each value proposition (3.1a) we generate the fact:

$$knows(l^{init}, 0, 0, 0). \tag{T2}$$

For each initial state constraint (3.1b) $\mathcal{C} \in \mathcal{ISC}$ such that $\mathcal{C} = \{l_1^{isc} \dots l_n^{isc}\}$ we iterate over each literal $l_i^{isc} \in \mathcal{C}$ and define $\{l_{i_1}^+, \dots, l_{i_m}^+\} = \mathcal{C} \backslash l_i^{isc}$ as the subset of literals $\mathcal{C}$ except l_i^{isc}. Then, for each $l_i^{isc} \in \mathcal{C}$ we generate the LP rule:

$$knows(l_i^{isc}, 0, 0, 0) \leftarrow \quad knows(\overline{l_{i_1}^+}, 0, 0, 0), \dots, knows(\overline{l_{i_m}^+}, 0, 0, 0). \tag{T3a}$$

$$\begin{aligned} &knows(\overline{l_{i_1}^+}, 0, 0, 0) \leftarrow knows(l_i^{isc}, 0, 0, 0). \quad \dots \\ &knows(\overline{l_{i_m}^+}, 0, 0, 0) \leftarrow knows(l_i^{isc}, 0, 0, 0). \end{aligned} \tag{T3b}$$

(T3a) denotes that if all literals except l_i^{isc} are known not to hold, then l_i^{isc} must hold. Rules (T3b) represent that if one literal l_i^{isc} is known to hold, then all others do not hold. At this stage of our work we only support constraints for the the initial state, because this is the only state in which they do not interfere with the postdiction rules. More general Static Causal Laws (Turner, 1999) as e.g. in the action language $\mathcal{C}$+ (Giunchiglia et al., 2004) would affect postdiction and causation rules. Their implementation is not trivial and therefore we leave this open to future work.

Actions $(\mathcal{A} \overset{\textbf{T4–T7}}{\longmapsto} \Gamma_{act})$

The generation of rules representing actions covers executability conditions, knowledge-level effects, and knowledge propositions.

Executability Conditions.
These reflect what an agent must know to execute an action. Let $\mathcal{EXC}^a$ of the form (3.1e) be the executability condition of action a in $\mathcal{D}$. Then LP($\mathcal{D}$) contains the following constraints:

$$\begin{aligned} &\leftarrow occ(a, N, B), not\ knows(l_1^{ex}, N, N, B). \qquad \ldots \\ &\leftarrow occ(a, N, B), not\ knows(l_{n_{ex}}^{ex}, N, N, B). \end{aligned} \tag{T4}$$

Effect Propositions (EP).
For every effect proposition $ep \in \mathcal{EP}^a$, of the form `(if (and` $l_1^c \ldots l_{n_c}^c$`) then` l^e`)`, LP($\mathcal{D}$) contains (T5), where $hasCond$ represents condition literals, $hasEff$ represents effect literals and $hasEP$ assigns an effect proposition to an action:

$$\begin{aligned} &hasEP(a, ep). \\ &hasEff(ep, l^e). \\ &hasCond(ep, l_1^c). \quad \ldots \quad hasCond(ep, l_{nc}^c). \end{aligned} \tag{T5}$$

Knowledge Level Effects of Non-Sensing Actions.
Knowledge-level effects represent knowledge gain by causation and postdiction, i.e. they reflect inference mechanisms **IM.3 – IM.5** of the operational semantics.

$$\begin{aligned} kCause(l^e, T+1, N, B) \leftarrow & apply(ep, T, B), N > T, \\ & knows(l_1^c, T, N, B), \ldots, knows(l_k^c, T, N, B). \end{aligned} \tag{T6a}$$

$$\begin{aligned} kPosPost(l_i^c, T, N, B) \leftarrow & apply(ep, T, B), \\ & knows(l^e, T+1, N, B), knows(\overline{l^e}, T, N, B). \end{aligned} \tag{T6b}$$

$$\begin{aligned} kNegPost(\overline{l_i^{c-}}, T, N, B) \leftarrow & apply(ep, T, B), knows(\overline{l^e}, T+1, N, B), \\ & knows(l_{i_1}^{c+}, T, N, B), \ldots, knows(l_{i_k}^{c+}, T, N, B). \end{aligned} \tag{T6c}$$

► *Causation* (T6a). This refers to Inference Mechanism **IM.3** (3.13). After an arbitrary number of steps n, if all condition literals l_i^c of an EP (3.1c) are known to hold at t, and if the action is applied at t, then at step $n > t$, it is known that its effect l^e holds at $t+1$. This is denoted by the atom $kCause(l^e, t+1, n, b)$ which reads as "by causation inference it is known that at step n in branch b literal l^e is known to hold at step $t+1$". The atom apply(ep,t,b) represents that action a with the EP ep happens at t in b.

► *Positive postdiction* (T6b). In the operational semantics, positive postdiction is defined as Inference Mechanism **IM.4** (3.14). In the ASP formulation, we iterate over

condition literals $l_i^c \in \{l_1^c, \dots, l_{n_c}^c\}$ of an effect proposition ep and add a rule (T6b) to the LP for each condition literal l_i^c. This defines how knowledge about the condition of an effect proposition is postdicted by knowing that the effect holds after the action but did not hold before. For example, if at n in b it is known that the complement $\overline{l^e}$ of an effect literal of an EP holds (i.e., $knows(\overline{l^e}, t, n, b)$), and if the EP is applied at t, and if it is known that the effect literal holds at $t+1$ ($knows(l^e, t+1, n, b)$), then the application of the EP must have produced the effect. Therefore one can conclude that the conditions $\{l_1^c, \dots, l_{n_c}^c\}$ of the EP must hold at t. This is represented by the atom $kPosPost(\overline{l_i^{c-}}, t, n, b)$ which reads as "by positive postdiction it is known that at step n in branch b literal $\overline{l_i^{c-}}$ is known to hold at step t".

► *Negative postdiction* (T6c). Negative Postdiction is defined as Inference Mechanism **IM.5** 3.15) in the operational semantics. We iterate over each potentially unknown condition literal $l_i^{c-} \in \{l_1^c, \dots, l_{nc}^c\}$ of an effect proposition ep. For each literal l_i^{c-}, we add one rule (T6c) to the program, where $\{l_{i_1}^{c+}, \dots, l_{i_{nc}}^{c+}\} = \{l_1^c, \dots, l_{nc}^c\} \backslash l_i^{c-}$ are the condition literals that are known to hold.
An atom $kNegPost(\overline{l_i^{c-}}, t, n, b)$ denotes that "by negative postdiction it is known that at step n in branch b literal $\overline{l_i^{c-}}$ is known to hold at step t". This covers the case where we postdict that a condition must be false if the effect is *known not to hold* after the action and all other conditions are known to hold. For example, if at n it is known that the complement of an effect literal l holds at some $t+1$, and if the EP is applied at t, and if it is known that all condition literals hold at t, except one literal l_i^{c-} for which it is unknown whether it holds. Then the complement of l_i^{c-} must hold because otherwise the effect literal would hold at $t+1$.

Knowledge Propositions.
We assign a knowledge proposition (KP) (3.1d) to an action a using a $hasKP$ predicate:

$$hasKP(a, f). \tag{T7}$$

Example 4.2 demonstrates how translation rules (T4) – (T7) generate the Logic Programming rules for Γ_{act}.

Goals ($\mathcal{G} \xrightarrow{\textbf{T8}} \Gamma_{goal}$)

For literals $l_1^{sg}, ..., l_{n_{sg}}^{sg} \in \mathcal{G}^{strong}$ in a strong goal proposition and $l_1^{wg}, ..., l_{n_{wg}}^{wg} \in \mathcal{G}^{weak}$ in a weak goal proposition we write the facts:

$$wGoal(l_1^{wg}). \quad \dots \quad wGoal(l_{n_{wg}}^{wg}). \tag{T8a}$$

$$sGoal(l_1^{sg}). \quad \dots \quad sGoal(l_{n_{sg}}^{sg}). \tag{T8b}$$

Example 4.2 Generating Γ_{act}
Recall the following specification of the action `drive`:

```
(:action drive  :executable ¬in_room
                :effect if is_open then in_room)
```

(T4) generates the executability constraint:

$$\leftarrow \mathit{occ}(\texttt{drive}, N, B), \mathit{not}\ \mathit{knows}(\texttt{in_room}, N, N, B).$$

The effect propositions are defined by (T5) as follows:

$$\begin{aligned}
&\mathit{hasEP}(\texttt{drive}, \texttt{ep_drive_0}).\\
&\mathit{hasEff}(\texttt{ep_drive_0}, \texttt{in_room}).\\
&\mathit{hasPC}(\texttt{ep_drive_0}, \texttt{open}).
\end{aligned}$$

where `ep_drive_0` is a syntactically generated label for the 0-th effect proposition of the `drive` action. The knowledge-level effects of the action are generated through (T6a–T6c):

$$\begin{aligned}
\mathit{kCause}(\texttt{in_room}, T+1, N, B) \leftarrow\ & \mathit{apply}(\texttt{ep_drive_0}, T, B), N > T,\\
& \mathit{knows}(\texttt{open}, T, N, B).\\
\mathit{kPosPost}(\texttt{open}, T, N, B) \leftarrow\ & \mathit{apply}(\texttt{ep_drive_0}, T, B),\\
& \mathit{knows}(\texttt{in_room}, T+1, N, B),\\
& \mathit{knows}(\neg\texttt{in_room}, T, N, B).\\
\mathit{kNegPost}(\neg\texttt{open}, T, N, B) \leftarrow\ & \mathit{apply}(\texttt{ep_drive_0}, T, B),\\
& \mathit{knows}(\neg\texttt{in_room}, T+1, N, B).
\end{aligned}$$

The first rule refers to *causation* (T6a): If it is known that the door is open, then it is known that the robot arrives in the living room after driving. The second rule represents *positive postdiction* (T6b): If it is known that the robot arrived in the living room, then it must be true that the door was open while it was driving. The third rule captures *negative postdiction* (T6c): If the robot did not arrive in the living room, then the door must have been closed.

These are used to trigger integrity constraints (F6a) and (F6d) of the domain independent theory which rule out Stable Models where the goals are not achieved.

4.4. Γ_{hpx} – Foundational Theory (F1) – (F7)

The foundational domain independent $\mathcal{HPX}$-theory covers auxiliaries (F1), concurrency (F2), inertia (F3), inference mechanisms (F4), sensing (F5), plan verification (F6) as well as plan generation and optimization (F7).

F1. Preliminaries and Auxiliary Definitions (Γ_{aux}) First, the maximal plan length and width is defined by instantiating atoms for steps (s) and branches (br) (F1a). Here, `maxS` and `maxBr` are constants of which the value is passed to the Logic Program at execution time.

$$\begin{aligned} &s(0..\texttt{maxS}).\\ &br(0..\texttt{maxBr}).\end{aligned} \tag{F1a}$$

To speed up the solving process we precompute inequality of branch labels with (F1b).

$$neq(B_1, B_2) \leftarrow B_1 \neq B_2, br(B_1), br(B_2). \tag{F1b}$$

In (F1c) we declare fluents f and their negations $neg(f)$ as literals, and we define an auxiliary predicate to denote the complement of literals.[1]

$$\begin{aligned} literal(neg(F)) &\leftarrow fluent(F).\\ literal(F) &\leftarrow fluent(F).\\ complement(neg(F), F) &\leftarrow fluent(F).\\ complement(L_1, L_2) &\leftarrow complement(L_2, L_1).\end{aligned} \tag{F1c}$$

F2. Concurrency (Γ_{conc}) Concurrency and the abstraction of effect propositions from actions is implemented as follows:

$$apply(EP, N, B) \leftarrow hasEP(A, EP), occ(A, N, B). \tag{F2a}$$

$$\begin{aligned} \leftarrow apply(EP_1, T, B), hasEff(EP_1, L), apply(EP_2, T, B), hasEff(EP_2, L),\\ EP_1 \neq EP_2, br(B), literal(L).\end{aligned} \tag{F2b}$$

(F2a) applies all effect propositions of an action a if that action occurs. (2) is a restriction concerning the application of similar effect propositions: two effect propositions are similar if they have the same effect literal. They may not be applied concurrently because otherwise the positive postdiction rule (T6b) and the inertia law (F3) would not work correctly. This is discussed in Section 7.1.

[1] Recall, that we often write $\neg f$ as a shorthand for $neg(f)$ to denote the negation of a fluent.

F3. Inertia (Γ_{in}) Inertia is applied in both forward and backward direction similar to (Gelfond and Lifschitz, 1993). To formalize this, we need a notion on knowing that a literal is *not* set by an action. This is expressed with the predicate $kNotSet$.

$$kNotSet(L,T,N,B) \leftarrow not\ kMaySet(L,T,B), uBr(N,B), s(T), literal(L). \quad \text{(F3a)}$$

$$kMaySet(L,T,B) \leftarrow apply(EP,T,B), hasEff(EP,L) \quad \text{(F3b)}$$

$$kNotSet(L,T,N,B) \leftarrow apply(EP,T,B), hasCond(EP,L'), hasEff(EP,L), knows(\overline{L'},T,N,B), complement(L',\overline{L'}), N >= T. \quad \text{(F3c)}$$

A literal could be known to be not set for two reasons: (1) if no effect proposition with the respective effect literal is applied, then this fluent can not be initiated. $kMaySet(l,t,b)$ (F3b) represents that at t an EP with the effect literal l is applied in branch b. If $kMaySet(l,t,b)$ does not hold then l is known not to be set at t in b (F3a). (2) a literal is known not to be set if an effect proposition with that literal is applied, but one of its conditions is known not to hold (F3c). Note that this requires the concurrency restriction (2), because without that restriction a literal could still be set by another effect proposition. We can now formulate forward inertia (F3d) and backward inertia (F3e) as follows:

$$knows(L,T,N,B) \leftarrow knows(L,T-1,N,B), kNotSet(\overline{L},T-1,N,B), complement(L,\overline{L}), T \leq N. \quad \text{(F3d)}$$

$$knows(L,T,N,B) \leftarrow knows(L,T+1,N,B), kNotSet(L,T,N,B), N > T. \quad \text{(F3e)}$$

Inertia is represented as inference mechanisms **IM.1** and **IM.2** (3.11–3.12) in the operational semantics.

The above inertia rules refer to a mental operation of inferring the temporal propagation of facts within an agent's knowledge. However, the operational semantics of $\mathcal{HPX}$ implies that knowledge itself is inertial (see Lemma B.8). That is, if at n it is known that l holds at t, then this is also known at $n+1$. Inertia of knowledge implemented as follows:

$$knows(L,T,N,B) \leftarrow knows(L,T,N-1,B), N \leq \texttt{maxS}. \quad \text{(F3f)}$$

F4. Inference Mechanisms (Γ_{infer}) The following rules are required to transfer the result of causation and positive and negative postdiction to knowledge.

$$knows(L,T,N,B) \leftarrow kCause(L,T,N,B). \quad \text{(F4a)}$$

$$knows(L,T,N,B) \leftarrow kPosPost(L,T,N,B). \quad \text{(F4b)}$$

$$knows(L,T,N,B) \leftarrow kNegPost(L,T,N,B). \quad \text{(F4c)}$$

F5. Sensing and Branching (Γ_{sense}) If sensing occurs, then each possible outcome of the sensing is assigned to a different branch, where $uBr(n, b)$ denotes that branch b is used at step n. Initially, only branch 0 is used (F5a).
$sNextBr$ (F5b) is an auxiliary predicate to denote that sensing produced a child branch. This is used in (F5c), which denotes that if no sensing result was produced, then a branch that was used in the past (at $n-1$) is used now (at n). If sensing did produce a child branch, i.e. if $sNextBr(n, b)$ is triggered, then $uBr(n, b')$ will be set by (F5j), where b' is the child branch's label.

$$uBr(0, 0). \tag{F5a}$$
$$sNextBr(N, B) \leftarrow sRes(L, N, B, B'). \tag{F5b}$$
$$uBr(N, B) \leftarrow uBr(N-1, B), not\ sNextBr(N-1, B), s(N). \tag{F5c}$$

An auxiliary predicate kw (F5d),(F5e) is an abbreviation for *knowing whether*:

$$kw(F, T, N, B) \leftarrow knows(F, T, N, B). \tag{F5d}$$
$$kw(F, T, N, B) \leftarrow knows(\neg F, T, N, B). \tag{F5e}$$

Next we describe the key rules that generate the sensing results, i.e. the $sRes$ atoms: atoms $occ(a, n, b), hasKP(a, f)$ denote that a sensing action with the knowledge proposition f occurs at step n in branch b. Sensing generates two branches. The positive sensing result is assigned to the original branch via (F5f). However, the sensing result is only generated if the negative sensing result is not already known to hold.
For the negative result, the choice rule (F5g) "picks" a valid child branch. It must be restricted that two sensing actions which occur at the same step n but in different branches b pick the same child branch (F5h). It must further be restricted that the negative sensing result is not assigned to already used branches (F5i).

$$sRes(F, N, B, B) \leftarrow occ(A, N, B), hasKP(A, F), not\ kw(neg(F), N, N, B). \tag{F5f}$$
$$1\{sRes(neg(F), N, B, B') : neq(B, B')\}1 \leftarrow occ(A, N, B), hasKP(A, F), not\ kw(F, N, N, B). \tag{F5g}$$

$$\leftarrow 2\{sRes(L, N, B, B') : br(B) : literal(L)\}, br(B'), step(N). \tag{F5h}$$
$$\leftarrow sRes(L, N, B, B'), uBr(N, B'), literal(L), neq(B, B'). \tag{F5i}$$

If an $sRes$ atom is produced, then the assigned branch is marked as used (F5j). Sensing results affect knowledge through (F5k). Note that like in the $sense$ function (3.8) of the operational semantics, sensing yields the value of a fluent at the time it was changed, i.e. at $N-1$ and not at N.

Finally, we need to implement the $\mathcal{HPX}$-restriction that two fluents can not be sensed concurrently (see 3.8). This is done with (F5l).

$$uBr(N, B') \leftarrow sRes(L, N-1, B, B'), s(N). \tag{F5j}$$
$$knows(L, N-1, N, B') \leftarrow sRes(L, N-1, B, B'), s(N). \tag{F5k}$$
$$\leftarrow 2\{occ(A, N, B) : hasKP(A, _)\}, br(B), s(N). \tag{F5l}$$

In order to to apply postdiction, causation and inertia rules to a child branch resulting from a sensing action, the child branch has to inherit knowledge (F5m) and application of EPs (F5n) from the parent branch.

$$knows(L, T, N, B') \leftarrow sRes(_, N-1, B, B'), neq(B, B'), knows(L, T, N-1, B), N \geq T. \tag{F5m}$$
$$apply(EP, T, B') \leftarrow sRes(_, N, B, B'), neq(B, B'), apply(EP, T, B), N \geq T. \tag{F5n}$$

F6. Plan Verification (Γ_{verify}) The ASP formalization supports both weak and strong goals. For weak goals there must exist one leaf where all goal literals are achieved and for strong goals the goal literals must be achieved in all leafs. Weak or strong goals are declared with the $wGoal$ and $sGoal$ predicates and defined through translation rules (T8). (F6a) defines atoms $notWG(n, b)$ which denote that a weak goal is not achieved at step n in branch b. An atom $allWGAchieved(N)$ reflects whether all weak goals are achieved at a step N (F6b). If they are not achieved at step `maxS`, then a corresponding model is not stable (F6c).

$$notWG(N, B) \leftarrow wGoal(L), uBr(N, B), not\ knows(L, N, N, B), literal(L). \tag{F6a}$$
$$allWGAchieved(N) \leftarrow not\ notWG(N, B), uBr(N, B). \tag{F6b}$$
$$\leftarrow not\ allWGAchieved(\texttt{maxS}). \tag{F6c}$$

Similarly, $notSG(n, b)$ denotes that a strong goal is not achived at step n in branch b (F6d). In contrast to weak goals, strong goals must be achieved in all used branches at the final step `maxS` (F6e).

$$notSG(N, B) \leftarrow sGoal(L), uBr(N, B), not\ knows(L, N, N, B), literal(L). \tag{F6d}$$
$$\leftarrow notSG(\texttt{maxS}, B), uBr(\texttt{maxS}, B). \tag{F6e}$$

Information about nodes where goals are not yet achieved is also generated. This is used in the plan generation part for pruning (F7a)–(F7b).

$$\begin{aligned} notGoal(N,B) &\leftarrow not SG(N,B). \\ notGoal(N,B) &\leftarrow not WG(N,B). \end{aligned} \tag{F6f}$$

F7. Plan Generation and Optimization (Γ_{plan}) In the generation part of the Logic Program, (F7a) and (F7b) implement sequential and concurrent planning respectively: for concurrent planning the choice rule's upper bound "1" is simply removed.[2] As described in Section 2.2.4, choice rules are used to "generate" atoms and hence can be interpreted as those mechanisms which span up the search tree. Optimal plans in terms of the number of actions are generated with the optimization statement (F7c), at the cost that computational complexity of the ASP solving raises from NP-completeness to Δ_2^P-completeness (see e.g. (Gebser et al., 2012b)).

$$1\{occ(A,N,B) : a(A)\}1 \leftarrow uBr(N,B), notGoal(N,B), N < \texttt{maxS}. \tag{F7a}$$

$$1\{occ(A,N,B) : a(A)\} \leftarrow uBr(N,B), notGoal(N,B), N < \texttt{maxS}. \tag{F7b}$$

$$\#minimize\{occ(_,_,_)@1\}. \tag{F7c}$$

4.5. Plan Extraction from Stable Models

To formally define how concurrent conditional plans (CCP) relate to Stable Models, we define a function $trans$ that takes a set of atoms S and two integer numbers $0 \leq n \leq \texttt{maxS}$, $0 \leq b \leq \texttt{maxB}$ as input and produces a CCP as output. n and b describe the position of the plan's root node in the transition tree, i.e. $trans(S,0,0)$ yields the conditional plan starting at the initial state.

$$trans(S,n,b) = \begin{cases} [] & \text{if } n = \texttt{maxS} \\ [[a_1||\ldots||a_m]; trans(S,n+1,b)] & \text{if } \neg\exists b', f : sRes(\neg f,n,b,b') \in S \\ [[a_1||\ldots||a_m]; \texttt{if } \neg f & \text{if } \exists b', f : sRes(\neg f,n,b,b') \in S \\ \quad \texttt{then } trans(S,n+1,b') & \\ \quad \texttt{else } trans(S,n+1,b)] & \end{cases} \tag{4.2}$$

where $\{a_1,\ldots,a_m\} = \{a | occ(a,n,b) \in \boldsymbol{P}\}$ and $\texttt{maxS}$ is a constant that limits the plan depth.

[2] In an actual implementation the LP may of course only contain one of these two choice rules, depending on which kind of planning is desired.

4.6. Relation between the ASP Implementation and the Operational $\mathcal{HPX}$-Semantics

The translation function (4.2) relates the occurrence of actions in the ASP implementation of $\mathcal{HPX}$ to state transitions in the operational semantics but it does not guarantee that the epistemic effects of actions are equivalent. In the following we formalize this equivalence and present a soundness Theorem.
As a prerequisite we define the auxiliary function (4.3) which describe a parent-child-relation between nodes in the transition tree. Let S be a Stable Model and $0 \leq n \leq \texttt{maxS}$, $0 \leq b \leq \texttt{maxB}$, $0 \leq b' \leq \texttt{maxB}$ be integers which are used to represent nodes in the transition tree. Then a function which defines whether a branch b' is a child branch of a node $\langle n, b\rangle$ wrt. a set of atoms S is defined as follows:

$$hasChild(n,b,b',S) \begin{cases} true & \text{if } \exists l : sRes(l,n,b,b') \in S \\ true & \text{if } b = b' \wedge \neg\exists l : sRes(l,n,b,b') \in S \\ false & \text{otherwise} \end{cases} \tag{4.3}$$

This is used to define the following ancestor relation:

$$ancestor(n_1,b_1,n_2,b_2,S) = \begin{cases} true & \text{if } \exists n,b : (ancestor(n_1,b_1,n,b,S) \wedge \\ & \qquad hasChild(n,b,b_2,S) \wedge n_2 = n+1) \\ false & \text{otherwise} \end{cases} \tag{4.4}$$

We are now ready to define the following auxiliary functions (4.5) which are used to map atoms in a Stable Model to nodes in the transition tree of the operational semantics:

$$\boldsymbol{\kappa}(n,b,S) = \{\langle l,t\rangle \,|knows(l,t,n,b) \in S\} \tag{4.5a}$$
$$\boldsymbol{\alpha}(n,b,S) = \{\langle a,t\rangle \,|\exists b',t : (occ(a,t,b') \in S \wedge ancestor(t,b',n,b,S))\} \tag{4.5b}$$
$$\mathfrak{h}(n,b,S) = \langle \boldsymbol{\alpha}(n,b,S), \boldsymbol{\kappa}(n,b,S)\rangle \tag{4.5c}$$
$$\boldsymbol{\epsilon}(n,b,S) = \boldsymbol{\epsilon}(\mathfrak{h}(n,b,S)) \tag{4.5d}$$

We also presume the following Definition 4.1 as a notational convention to formally describe the relation between the ASP formalization and the operational semantics.

Definition 4.1 (Notation to relate ASP implementation with $\mathcal{HPX}$ semantics)

- *$\texttt{maxS}$ and $\texttt{maxB}$ are constants denoting the maximal plan depth and width respectively. $0 \leq n \leq \texttt{maxS}$, $0 \leq b \leq \texttt{maxB}$ and $0 \leq b' \leq \texttt{maxB}$ denote variables for steps and branches respectively.*

- *$\mathcal{D}$ is a domain description with the initial h-state $\mathfrak{h}_0$*

- *$LP(\mathcal{D})$ is the Logic Program of a domain description $\mathcal{D}$ without the plan-generation rule (F7) and without the goal statements generated by translation rule (T8)*
- *$S_{\mathcal{D}}^{\boldsymbol{P}}$ is a Stable Model of $LP(\mathcal{D}) \cup \boldsymbol{P}$ where $\boldsymbol{P}$ is a set of $occ(a,n,b)$ atoms with $0 \leq n <$ `maxS` such that*
  - $\forall a,n,b : \big(occ(a,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n,b)\big)$.[3]
  - $\forall n,b : \big(uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \exists a : occ(a,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)$ [4]
- *$\boldsymbol{A}_{n,b} = \{a | occ(a,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\}$ is a set of actions applied at a transition tree node with the "coordinates" $\langle n,b \rangle$.*

Functions (4.3) and (4.5) along with Definition 4.1 allow us to provide a complete summary of how ASP atoms relate to the operational $\mathcal{HPX}$ semantics in Table 4.1. Theorem 4.1 is the core soundness theorem concerning knowledge:

Theorem 4.1 (Soundness of ASP Formalization of $\mathcal{HPX}$) *For all l,n,t,b: if there exists a b' such that $hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}})$ and $knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$, then there exists an h-state $\mathfrak{h} \in \Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))$ such that $\mathfrak{h} \models \langle l,t \rangle$.*

Proof: The theorem follows directly from Lemma A.2 in Appendix A.
The Lemmata which we mention in Table 4.1 are defined and proven in Appendix A. The results concern the *soundness* of the ASP implementation wrt. the operational semantics. *Completeness* results are not provided because the ASP implementation is incomplete wrt. the operational semantics. This is discussed in Section 7.1. In addition, to demonstrate that the ASP implementation provides results for many problem instances, we present a number of examples throughout this thesis where the ASP formalization correctly generates knowledge (see e.g. Chapter 6).

[3] This restriction reflects the mechanics of the plan generation rule (F7) which only generates $occ(a,n,b)$ atoms if $uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

[4] This restricts that there are no "gaps" in a plan, i.e. for all nodes in used branches there occurs at least one action. Note that this restriction is met for all $occ/3$ atoms which are generated by the plan generation rule (F7).

	ASP formalization	**Operational Semantics**	
Core predicates			
Knowledge	$knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle l, t \rangle \in \boldsymbol{\kappa}(\mathfrak{h})$	Theo. 4.1
Inertia	$kNotSet(\bar{l}, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$inertial(l, t, \mathfrak{h})$	Lem. A.3
App. of EP	$apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle ep, t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h})$	Lem. A.7
Sensing	$sRes(l, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle l, n \rangle \in sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}))$	Lem. A.10
Action occ.	$occ(a, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$a \in \boldsymbol{A}_{n,b}$	Def. 4.1
Inference mechanisms			
Causation	$kCause(l, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle l, t \rangle \in add_{fwd}(\mathfrak{h})$	
Pos. post.	$kPosPost(l, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle l, t \rangle \in add_{pd^{pos}}(\mathfrak{h})$	
Neg. post.	$kNegPost(l, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\langle l, t \rangle \in add_{pd^{neg}}(\mathfrak{h})$	
	where b' such that $hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}})$ and $t \leq n$.	where $\mathfrak{h} \in \Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})$ and $t \leq n$.	
Auxiliary predicates			
Eff. prop.	$hasEP(a, ep) \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$ep \in \mathcal{EP}^a$	Lem. A.12
Eff. literal	$hasEff(ep, f) \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$e(ep) = f$	Lem. A.12
Cond. lit.	$hasCond(ep, f) \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$f \in c(ep)$	Lem. A.12
Know. prop.	$hasKP(a, f) \in S_{\mathcal{D}}^{\boldsymbol{P}}$	$\mathcal{KP}^a = f$	Lem. A.12

Table 4.1.: Relation between ASP formalization of $\mathcal{HPX}$ and its operational semantics

5 An $\mathcal{HPX}$ Online Planning Framework

The basic h-approximation formalism is designed for offline problem solving. This means that a conditional plan is generated and it is checked whether redefined goals are entailed in future world states. After the generation of the plan an agent executes the plan.
However, in practice it is often useful to *interleave* planning and plan execution. Therefore we integrate the $\mathcal{HPX}$ implementation in an online system architecture for general Cognitive Robotics control (Section 5.1).
To this end, we extend the $\mathcal{HPX}$ implementation so that it is capable of interleaving planning and plan execution and we combine this with *abductive explanation* (Section 5.2). We also present modifications which improve the computational performance of the problem solving and implement *typing* to add expressiveness to the PDDL-like input language.

5.1. System Architecture

The $\mathcal{HPX}$ compiler which translates the PDDL-like input language into Logic Programming Rules is combined with a controller and the incremental online ASP solver *oclingo* (Gebser et al., 2011a) to constitute a Cognitive Robotic control system. This is illustrated in Figure 5.1. The controller communicates new goals, sensing results and execution statements to the solver and is also responsible for the plan execution and the communication with actuators and sensors.

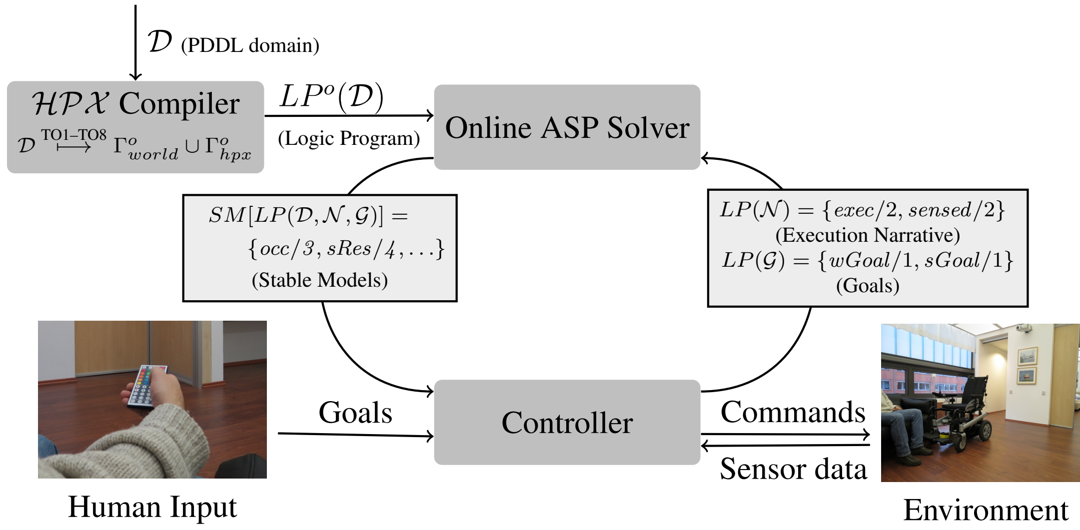


Figure 5.1.: Architecture of the online planning framework

In the online architecture, the Logic Program to be solved is given as

$$LP(\mathcal{D}, \mathcal{G}, \mathcal{N}) = LP^o(\mathcal{D}) \cup LP(\mathcal{G}) \cup LP(\mathcal{N}) \tag{5.1}$$

where

- $LP^o(\mathcal{D}) = \Gamma^o_{hpx} \cup \Gamma^o_{world}$
  - Γ^o_{hpx} is an online version of the domain independent theory, constituted by the Logic Programming rules (FO1) – (FO9). Rules (FO1) – (FO7) are online versions of their corresponding offline domain independent rules (F1) – (F7) which we describe in Chapter 4. (FO8) – (FO9) are additional rules which cover the physical execution of actions and abductive explanation.
  - Γ^o_{world} is an online version of the domain specific part of the extended $\mathcal{HPX}$ implementation. Γ^o_{world} is generated by applying translation rules (TO1) – (TO8) on the PDDL-like domain description. (TO1) – (TO8) are online versions of their corresponding offline counterparts (T1) – (T8) which we describe in Chapter 4.
- $LP(\mathcal{G})$ denotes a set of dynamically stated goals through $wGoal/1$ and $sGoal/1$ atoms.
- $LP(\mathcal{N})$ denotes an *execution narrative*. $LP(\mathcal{N})$ is a set of $exec/2$ and $sensed/2$ atoms which reflect which actions were executed and which sensing results were obtained.

Once a Stable Model (i.e. a plan) $\boldsymbol{P} \in SM[LP(\mathcal{D},\mathcal{G},\mathcal{N})]$ is found, it is sent to the controller which starts to execute the plan. During execution, the controller reports the execution narrative $LP(\mathcal{N})$ back to the solver. The solver adopts the search space according to this information and expands the plan accordingly. The updated Stable Models are thereupon sent to the controller again which executes the corresponding plan. The loop is repeated until the goal is achieved or the problem becomes unsolvable.
For illustrations we refer to Section 6.2 where we present an elaborate use case that depicts the functioning of the online planning system in detail.

5.2. Extensions for Online Planning

In order to enable online planning we made some extensions to the original ASP-based $\mathcal{HPX}$ implementation. The main differences to the original formalism which we describe in Chapter 4 are that *(a)* online ASP solving allows to have an incremental planning horizon, *(b)* action planning and action execution is interleaved *(c)* abductive explanation is integrated in the framework *(d)* some basic performance optimizations are implemented and *(e)* *typing* is introduced to extend the expressiveness of the PDDL-like input syntax. Details concerning the implementation of the extensions are provided in Appendix D.3, including a Listing of the domain independent online theory Γ^o_{hpx}.

5.2.1. Incremental Planning Horizon Extension

If using non-incremental ASP solving for planning it is required to pre-set a fixed planning horizon before solving the LP. That is, the constant `maxS` which we introduced in Chapters 3 and 4 and which represents the plan length has to be defined before the planning starts. However, since the minimal plan length is usually unknown it can be problematic to find a suitable value for this constant.
A solution is incremental ASP solving which we describe in Section 2.2.8. This allows one to expand the planning horizon dynamically, to a minimal extend which is required to find a plan. At the same time this approach guarantees that generated plans are minimal in terms of the number of state transitions.

5.2.2. Interleaving Planning and Plan Execution

Rules (F1), (F2), (F3), (F6) and (T3) – (T8) require only the following minor modifications: the variable N is replaced by the iterator t, the keywords $\#base$, $\#cumulative$ and $\#volatile$ are placed appropriately to partition the incremental Logic Program into its respective parts. For example, recall the LP rule (F3e) for backward inertia:

$$knows(L,T,N,B) \leftarrow knows(L,T+1,N,B), kNotSet(L,T,N,B), N > T$$

In the online planning LP this rule is placed within the $\#cumulative$ part and N is replaced by t:

$$\begin{aligned}&\#cumulative\ \ t\\ &knows(L,T,t,B) \leftarrow \quad knows(L,T+1,t,B), kNotSet(L,T,t,B), t > T\end{aligned}$$

In addition to replacing N by t and splitting the program into its respective parts we replace the $apply/3$ predicates by $apply/4$. This is due to a restriction of the ASP solver *oclingo* which we are using. With $apply/3$ predicates it would happen that rules are re-grounded during the iterative problem solving. This is currently not allowed within *oclingo*. In addition, the keyword $\#external$ is used to mark the predicates $sensed/2$, $exec/2$, $wGoal/1$ and $sGoal/1$ as external. This is required for the ASP solver to consider that respective atoms may be added to the Logic Program on-the-fly.
In the following we describe the modifications to the translation rules (T4), (T6). Translation rules (TO3), (TO5), (TO7) and (TO8) are identical to their offline versions (T3), (T5), (T7) and (T8). Modifications to translation rules (T1), (T2) are described within the context of performance optimizations in Section 5.2.4.
We also describe necessary modifications to the domain-independent Logic Programming rules (F5), (F7), and we present two additional rules (FO8) – (FO9).

TO4 and TO6. Actions (Γ^{O}_{act})

We reformulate rule (T4) in that we simply replace the variable N by the iterator t:

$$\begin{aligned}&\leftarrow occ(a,t,B), not\ knows(l^{ex}_1,t,t,B), uBr(t,B). \qquad \ldots\\ &\leftarrow occ(a,t,B), not\ knows(l^{ex}_n,t,t,B), uBr(t,B).\end{aligned} \tag{TO4}$$

Similarly we rewrite rules (T6a) – (T6c) as follows:

$$\begin{aligned}knows(l^e,T+1,t,B) \leftarrow &apply(ep,T,t,B), t > T,\\ &knows(l^c_1,T,t,B),\ldots,knows(l^c_n,T,t,B).\end{aligned} \tag{TO6a}$$

$$\begin{aligned}knows(l^c_i,T,t,B) \leftarrow &apply(ep,T,t,B), uBr(t,B),\\ &knows(l^e,T+1,t,B), knows(\overline{l^e},T,t,B).\end{aligned} \tag{TO6b}$$

$$\begin{aligned}knows(\overline{l^{c-}_i},T,t,B) \leftarrow &apply(ep,T,t,B), knows(\overline{l^e},T+1,t,B), uBr(t,B),\\ &knows(l^{c+}_{i_1},T,t,B),\ldots,knows(l^{c+}_{in},T,t,B).\end{aligned} \tag{TO6c}$$

FO5. Sensing and Branching (Γ^o_{sense})

If a sensing result was obtained through physical execution of a sensing action, the planner must not consider branches of the execution tree anymore in which the contrary of the sensing result was presumed. We refer to this behavior as *commitment* to sensing results. That is, if a message $sensed(f, t)$ was received, then atoms representing the contrary sensing result $sRes(\neg f, t, b, b')$ may not be produced. We respect this by replacing rules (F5f) and (F5g) with the following rules (FO5e), (FO5f):

$$\begin{aligned} sRes(F, t-1, B, B) \leftarrow\ & occ(A, t-1, B), hasKP(A, F), \\ & not\ sensed(neg(F), t-1), not\ kw(F, t-1, t-1, B). \end{aligned} \tag{FO5e}$$

$$\begin{aligned} 1\{sRes(neg(F), t-1, B, B') : neq(B, B')\}1 \leftarrow\ & occ(A, t-1, B), hasKP(A, F), \\ & not\ sensed(F, t-1), not\ kw(F, t-1, t-1, B). \end{aligned} \tag{FO5f}$$

The difference to the original rules (F5f) and (F5g) are the $not\ sensed(neg(F), t-1)$ (resp. $not\ sensed(F, t-1)$) atoms in the rules' bodies which prevent from generating child branches if a contradictory sensing result was obtained. In rules (FO5e) – (FO5f) one would usually use a parameter t instead of $t-1$, but we discovered that the online solver *oclingo* produces an error message if we replace $t-1$ by t. This is probably caused by restrictions within *oclingo* v3.0.92 which is a beta version.

FO7. Plan Generation (Γ^o_{plan})

For the plan generation part of the online theory, we need to prevent rules (F7) from proposing additional action occurrences at steps that were already executed in the past. For this reason, we replace the rule for sequential offline planning (F7a)[1] with the following extended rule:

$$\begin{aligned} 1\{occ(A, t, B) : action(A)\}1 \leftarrow \\ uBr(t, B), not\ executedStep(t), notGoal(t, B). \end{aligned} \tag{FO7a}$$

A new component in (FO7a) is $not\ executedStep(t)$ which denotes that an action can only be planned for at a step t if the step has not already been executed. This is defined in (FO8b) .

FO8. Execution (Γ^o_{exec})

The interleaving of action planning and execution requires that the search space generated by the planner does not contradict the real action execution. For instance, if the action of opening a door was executed at a step t, then the planner always has to consider the

[1] The same is done respectively for the concurrent planning rule (F7b).

execution of this action in its search tree. Similar to commitment to sensing results, we refer to this behavior as *commitment to action execution* which is implemented by the following rule:

$$occ(A,t,B) \leftarrow exec(A,t), action(A), uBr(t,B). \tag{FO8a}$$

That is, whenever the execution of an action was reported by the controller (represented by $exec(A,t)$) the ASP solver considers the occurrence of this action in all nodes of the transition tree.
The following rules implement an abbreviation predicate $executedStep/1$ which denotes that a step t has already been physically executed.

$$\begin{aligned} executedStep(t) &\leftarrow exec(A,t), action(A). \\ executedStep(t) &\leftarrow sensed(L,t), literal(L). \end{aligned} \tag{FO8b}$$

In addition to rules (FO8a) and (FO8b) we need two more rules (FO8c) and (FO8d) which assure that sensed knowledge becomes actually known if acquired unexpectedly, i.e. without having anticipated the execution of a sensing action. This is useful for continuously *monitoring* world properties, such as the open-state of a door.

$$knows(L,t,t,B) \leftarrow sensed(L,t), uBr(t,B), literal(L). \tag{FO8c}$$

$$\leftarrow sensed(L_1,t), uBr(t,B), knows(L_2,t,t,B), complement(L_1,L_2). \tag{FO8d}$$

5.2.3. Exogenous Events and Abductive Explanation

In domains where world properties change unexpectedly, it is useful to monitor these properties continuously to make sure that their correct values are always known. For instance, one may open a door and then send a robot through the door, but one never knows whether the door is accidentally closed by another (human) agent after the door was opened. We call such unexpected actions which can not be controlled by the reasoning agent *exogenous actions*. In contrast, we call actions that are executed by the reasoning agent *endogenous actions*.
For instance, consider the following action description:

```
(:action closeDoorExo exogenous
  :effect ¬is_open)
```

This describes that a door can be closed by an external agent, syntactically indicated by the keyword exogenous. Note that it makes no sense to define executability conditions (3.1e) for exogenous actions, as these refer to the knowledge of the reasoning agent.[2] Hence we assume that all exogenous action do not have executability conditions.

[2]As an example consider two agents R1 and R2 which can move from room A to room B through a door.

FO9. Exogenous Events and Abductive Explanation (Γ^{o}_{exo})

To account for exogenous actions in the generated $\mathcal{HPX}$-Logic Program, we modify translation rule (T1) as follows: for every fluent f, endogenous action a or exogenous action a_{exo}, LP($\mathcal{D}$) contains the facts:[3]

$$\mathit{fluent}(f). \quad \mathit{action}(a). \quad \mathit{exoAction}(a_{exo}).$$

In our framework, unexpected change of world properties is modeled by *abductive explanation*. In Section 2.1.2 we argue that the abductive explanation problem is technically equal to the planning problem: given an initial state, one is interested in a course of actions that leads to a final state. In the context of abduction, the final state is any state that has a certain property which is to be explained. We implement the explanation mechanism with the following choice rule:

$$\begin{aligned}&0\{\mathit{exoHappened}(A, t-1, B)\\ &\quad : \mathit{hasEP}(A, EP) : \mathit{hasEff}(EP, \overline{L}) : \mathit{exoAction}(A)\}1 \leftarrow \\ &\quad\quad \mathit{sensed}(\overline{L}, t), \mathit{uBr}(t, B), \mathit{knows}(L, t-1, t, B), \mathit{complement}(L, \overline{L}).\end{aligned} \tag{FO9a}$$

The rule reflects that if at step t it is known that literal l holds at the previous step $t-1$, but it is sensed that at t the complement, $\bar{l}$ holds, then propose an exogenous action (denoted $\mathit{exoAction}(a)$) which could possibly have set l.
In addition to (FO9a) another rule (FO9b) that applies the effect propositions of exogenous actions is required:

$$\mathit{apply}(EP, t-1, t, B) \leftarrow \mathit{hasEP}(A, EP), \mathit{exoHappened}(A, t-1, B). \tag{FO9b}$$

Note that exogenous actions are only used for explanation if there occurred no endogenous action which may also have set the value of concern for the following reason: the ASP implementation of the h-approximation has the restriction that no two actions with the same effect literal may happen concurrently. Therefore, if an endogenous action with the respective effect literal has been executed, an exogenous action with the same effect literal will not be considered for explanation.
A problem with abductive explanation is that explanations for unexpected world change are often ambiguous. For example, if a door is closed exogenously and two persons could have closed it then without additional knowledge it is impossible to tell which one of the persons actually closed the door. This is discussed in Section 7.2.

Here it is sensible to use an executability condition which requires the agents to know whether the door is open before passing it. We consider R1 to be the reasoning agent and R2 to be an exogenous agent. Assume that the exogenous actions which R2 can execute are modeled within R1's domain description. Then from R1's perspective it does not make sense to represent the exogenous move-action of R2 with an executability condition concerning open-ness of the door. This would only require R1 to known whether the door is open, but the moving would refer to movement of R2. Therefore the executability conditions is semantically inadequate.

[3] Note that this rule is not the final version of the online translation rule TO1: in Section 5.2.4 we describe how the rule is extended further with a performance optimization statement.

5.2.4. Performance Optimization: Static Relations and Impossible Actions

Though practical performance is not the main focus in the development of $\mathcal{HPX}$, we implement an extension to reduce the computation time by considering so-called *static relations*. Static relations represent relations between objects which can not be changed, e.g. the connectivity of rooms. We extend the $\mathcal{HPX}$-compiler such that it automatically marks these fluents as static relations which do not occur in any effect proposition or knowledge proposition of an action. This is formalized in Definition 5.1.

Definition 5.1 (Static Relations) *For a domain $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, a fluent f is called* static relation *if the the following conditions hold:*

- $\forall a \in \mathcal{A} : \mathcal{KP}^a \neq f$
  (There exists no action a with a knowledge proposition $\mathcal{KP}^a = f$)
- $\forall a \in \mathcal{A} : \forall ep \in \mathcal{EP}^a : e(ep) \notin \{f, \neg f\} \wedge \{f, \neg f\} \cap c(ep) = \emptyset$
  (There exists no action a with an effect proposition ep that has a condition literal f or $\neg f$ or an effect literal f or $\neg f$.)

Obviously there is no need to represent static relations with the 4-ary $knows$ predicate, since they can not change; it is sufficient to use a 1-ary $holds$ predicate to denote whether or not a world property is (and stays) true or false. This simplifies the knowledge representation and results presented in Section 6.2 reveal that especially the grounding time of the ASP solving is reduced with this extension.
Static relations bring another advantage in combination with executability conditions in that they allow one to quickly determine whether or not an action is actually *impossible*. This can drastically reduce the search space, especially if many impossible actions are specified. For instance, driving from a room A to room B is impossible if there is no connection between room A and B, and in this case the action does not have to be considered.
We implement this thought by simply not instantiating actions which are impossible on the Logic Programming level. This is realized with the following extended version of translation rule (T1). For every fluent f, endogenous action a and exogenous action a_{exo}, $LP^o(\mathcal{D})$ contains the facts:

$$\begin{aligned} &fluent(f). \\ &action(a) \leftarrow holds(l_1^{ex}), \ldots, holds(l_{n_{exs}}^{ex}). \\ &exoAction(a_{exo}). \end{aligned} \tag{TO1}$$

where $l_1^{ex}, \ldots, l_{n_{exs}}^{ex}$ are these literals in the executability condition $\mathcal{EXP}^a$ of an action a which are static. Exogenous actions are not affected, because as stated in Section 5.2.2 these actions do not have executability conditions.

Rule (T2) which generates initial knowledge is modified such that $knows/4$ is replaced by $holds/1$ as follows: let `(:init (and` $l_1^{init} \dots l_{n_{in}}^{init}$`))` be a value proposition (3.1a). Then for each literal $l^{init} \in \{l_1^{init}, \dots, l_{n_{in}}^{init}\}$ we generate the fact

$$\begin{array}{ll} knows(l^{init}, 0, 0, 0). & \text{if } l^{init} = f \text{ or } l^{init} = \neg f \text{ and } f \text{ is not static} \\ holds(l^{init}). & \text{otherwise} \end{array} \tag{TO2}$$

Other translation rules do not have to be modified because by Definition 5.1 static relations do not occur in effect propositions or knowledge propositions of actions.
The result in terms of computational performance is investigated in Section 6.2.2 where we analyze a planning problem with and without accounting for static relations.

5.3. Incremental Reactive Planning

In Section 2.2 we illustrate that online ASP solving relies on the *module theory* (Oikarinen and Janhunen, 2006). However, in order to apply the theory some requirements have to be met to guarantee the *compositionality* of separate Logic Programming modules.
In order to show that incremental online $\mathcal{HPX}$-Logic Programs satisfy the compositionality conditions we investigate their semantics from this point of view.
According to Section 2.2.8, an incremental Logic Program is given as

$$R[t] = B \cup \bigcup_{0 \leq j \leq t} P[j] \cup Q[t] \tag{5.2}$$

for some $t \geq 0$ where B represents the $\#base$ part, P represents the $\#cumulative$ part and Q represents the $\#volatile$ part. These particular constituents are identified in detail in Appendix D.3.2.
As described in Section 2.2.8, one needs to consider certain restrictions on the $\#external$ input atoms wrt. the online Logic Program, i.e. the modularity condition described by Definition 2.16. In the $\mathcal{HPX}$-Logic Program, we have that $sensed/2$, $exec/2$, $wGoal/1$ and $sGoal/1$ atoms are external, and the controller sends inputs of the following form to the ASP solver:

```
#step t_max + 1.
sensed(l_s,t_s).
exec(a,t_e).
wGoal(l_wg).
sGoal(l_sg).
#endstep.
```

where $t_{max} = max(t_s, t_e)$ and $t_s, t_e \geq 0$. Intuitively, l_s are the atoms that are sensed, a are the actions that are executed, l_{wg} are weak goal statements and l_{sg} are strong goal statements.

Fortunately, *oclingo* checks automatically whether the modularity condition (see Definition 2.16) holds. We have not formally proven that all input sent by the controller satisfies the condition. However, *oclingo* internally performs a check whether the modularity condition is met. If the condition is not met, *oclingo* provides an error message and no solution is generated. Therefore, the correctness of our reasoning results is guaranteed. Furthermore, we have not observed that *oclingo* provides an error message during the conduction of our experiments described in Chapter 6.

5.4. Extending Expressiveness: Typing

Though the core focus of this work lies in the epistemic action-theoretic aspects, we also make some basic efforts to improve the practical applicability of the implemented system. To this end, we improve the expressiveness of our PDDL-like input language and implement *typing*. Typing allows one to define *action schemes* which are more general than the effect propositions (3.1c), knowledge propositions (3.1d) and executability conditions (3.1e) defined in Section 3.1. Typing is implemented in most PDDL dialects, for details we refer to (McDermott et al., 1998). As an example for a domain description which involves typing we refer to Appendix D.4 where we present the complete domain description of a use case described in Section 6.2.
In the following we describe four language elements, namely type definitions, predicate definitions, object definitions and action scheme definitions.
The following example Listings 5.1 – 5.4 refer to a domain specification where robotic agents (denoted `rolland1` and `rolland2`) can move through doors to navigate between rooms (denoted `corr1` ,`bed` ,`liv`, `office`, `bath`).

Type Definitions

Types are defined as follows:

$$\begin{array}{l}\texttt{(:types} \\ \qquad type_1 \ \texttt{-} \ type_1^{parent} \\ \qquad \vdots \\ \qquad type_{n_t} \ \texttt{-} \ type_{n_t}^{parent}\end{array} \tag{5.3a}$$

where $type$ are child-types of $type^{parent}$.

For instance, the following Listing 5.1 implements the types `Door`, `Room`, `Agent`, `Person` and `Robot`, where `Person` and `Robot` are sub-types of `Agent`.

```
(:types
  Door
  Room
  Agent
  Person - Agent
  Robot - Agent)
```

Listing 5.1: Type definitions in the extended input language

Object Definitions

Objects are defined as follows:

$$\begin{aligned} &\texttt{(:objects} \\ &\quad object_1 \texttt{ - } type_1 \\ &\quad \vdots \\ &\quad object_{n_o} \texttt{ - } type_{n_o}\texttt{)} \end{aligned} \tag{5.3b}$$

The following Listing 5.2 implements five rooms `corr1`, `bed`, `liv`, `office` and `bath`, three doors `d1`, `d2` and `d4`, two robots, `rolland` and `iwalker` and a person `paul`.

```
(:objects
  corr1,bed,liv,office,bath - Room
  d1,d2,d4 - Door
  rolland1, rolland2 - Robot
  paul - Person)
```

Listing 5.2: Object definitions in the extended input language

Predicate Definitions

Predicates are defined as follows:

$$\begin{aligned} &\texttt{(:predicates} \\ &\quad pred_1(type_1^1, \ldots, type_1^{t_1}) \\ &\quad \vdots \\ &\quad pred_{n_p}(type_{n_p}^1, \ldots, type_{n_p}^{t_{n_p}}) \\ &\texttt{)} \end{aligned} \tag{5.3c}$$

Predicate definitions are used to define the set of domain fluents $\mathcal{F}_{\mathcal{D}}$.[4] For instance, Listing 5.3 implements five predicates `hasDoor/2`, `connected/2`, `inRoom/2`, `open/1` and `abnormal_drive/1`.

```
(:predicates
  hasDoor(Room, Door)
  connected(Room, Room)
  inRoom(Agent, Room)
  open(Door)
  abnormal_drive(Robot))
```

Listing 5.3: Predicate definitions in the extended input language

In combination with the object and type definitions stated in Listings 5.1 and 5.2 the predicate `hasDoor/2` evaluates to 15 fluents `hasDoor(corr1,d1)`, `hasDoor(corr1,d2)`, ..., `hasDoor(bath,d2)`, `hasDoor(bath,d4)`, and similar for the other predicates.

Action Scheme Definitions

Action schemes are defined as follows:

$$\begin{aligned}&\texttt{(:action } act \texttt{ :parameters (}?v_1 \texttt{ - } type_1\texttt{)},\ldots,?v_{n_t} \texttt{ - } type_{n_t}\texttt{)}\\ &\qquad\texttt{:executable (and } fluentLitScheme_1,\ldots,fluentLitScheme_{n_{exc}}\texttt{)}\\ &\qquad\texttt{:effect (and } epScheme_1,\ldots,epScheme_{n_{ep}}\texttt{)}\\ &\qquad\texttt{:observe } fluentLitScheme^{obs}\texttt{)}\end{aligned} \tag{5.3d}$$

where

- $\texttt{:parameters (}?v_1 \texttt{ - } type_1\texttt{)},\ldots,?v_{n_t} \texttt{ - } type_{n_t}\texttt{)}$ is a parameters section where $?v_1,\ldots,?v_{n_t}$ denote variable names of the respective types.

  The parameters section is used to define variables which are used in *fluent literal schemes* denoted $fluentLitScheme$. Fluent literal schemes have the form $pred(arg_1,\ldots,arg_{n_p})$ where $pred$ is a predicate name and $arg_1,\ldots,arg_{n_p}$ are either variable names which must occur in the parameters section or objects defined in the objects definition (5.3b). Action scheme definitions are only valid if the types of variables defined in the parameters section and the objects definition coincides with the type assignment defined in the predicate definition (5.3c). In the following we assume that all action scheme definitions are valid.

[4] In the basic ASP iplementation of $\mathcal{HPX}$ we have assumed that the set of domain fluents $\mathcal{F}_{\mathcal{D}}$ is automatically extracted from the domain definition. However, in practice we define the set of predicates (and thereby the set of domain fluents) manually with the predicate definitions. This makes the domain design less error-prone because the domain designer has to think more carefully about the fluents he is using. Also, typing errors are reduced. Defining domain predicates manually is typical for PDDL-planning in general (see e.g. (McDermott et al., 1998)).

- `:executable (and` $\mathit{fluentLitScheme}_1, \ldots, \mathit{fluentLitScheme}_{n_{exc}}$`)` is an optional executability section.
- `:effect (and` $\mathit{epScheme}_1, \ldots, \mathit{epScheme}_{n_{ep}}$`)` is an optional effect section where $\mathit{epScheme}_1, \ldots, \mathit{epScheme}_{n_{ep}}$ denote effect proposition schemes of the form `if (and` $\mathit{fluentLitScheme}^c_1 \ldots \mathit{fluentLitScheme}^c_{n_c}$`) then` $\mathit{fluentLitScheme}^e$.
- `:observe` $\mathit{fluentLitScheme}^{obs}$ is an optional observation section.

All action scheme definitions must have a parameter section and an effect section or an observation section. As an example consider Listing 5.4 which models the action of a robot driving through a door and the sensing action of locating a robot.
Actions are generated from action schemes in the obvious way, i.e. by instantiating the action scheme with all possible combinations of parameters, according to the type and object definitions. For example, the object and type definitions stated in Listings 5.1 and 5.2 define 2 robots, 3 doors and 5 rooms. Hence, the action scheme `drive_door` results in $2 \cdot 3 \cdot 5 \cdot 5 = 150$ actions.[5]

```
(:action drive_door
  :parameters (?robo - Robot ?door - Door ?from ?to - Room)
  :executable (and
    open(?door)
    hasDoor(?from, ?door)
    hasDoor(?to, ?door)
    inRoom(?robo, ?from)
    !inRoom(?robo, ?to))
  :effect (and
    (if !abnormal_drive(?robo) then !inRoom(?robo, ?from))
    (if !abnormal_drive(?robo) then inRoom(?robo, ?to))))

(:action senseLocation
  :parameters (?robo - Robot ?room - Room)
  :observe inroom(?robo, ?room))
```

Listing 5.4: Action scheme definitions in the extended input language

[5] However, note that by applying the modified translation rule (TO1) described in Section 5.2.4 usually only a small subset of these actions are actually generated because “impossible” actions are rules out.

6 Application in a Smart Home and Evaluation

To evaluate the $\mathcal{HPX}$ formalism and integration in the online planning framework, we present two scenarios where $\mathcal{HPX}$ is used for action planning, abnormality detection and abductive explanation in the Bremen Ambient Assisted Living Lab (BAALL) (Krieg-Brückner et al., 2010). BAALL is a robotic Smart Home environment equipped with an autonomous robotic wheelchair and various actuators and sensors.
The first scenario (Section 6.1) has mainly the illustrative purpose of describing postdiction and other basic offline inference mechanisms using the basic offline ASP implementation of $\mathcal{HPX}$.
The second scenario (Section 6.2) illustrates how $\mathcal{HPX}$ is used for online planning and how abductive explanation is interleaved with action planning and plan execution. In order to assess the practical applicability of $\mathcal{HPX}$ in actual robotic environments, we also provide an empirical evaluation in terms of computation time for this scenario.
In addition to the case studies, and though computational performance is not the main focus of this work, we present an empirical evaluation in Section 6.3. We compare the computation time of the $\mathcal{HPX}$ planning system with the CFF planner (Hoffmann and Brafman, 2005) and the ASCP planning system (Tu et al., 2007) for three typical benchmark problems from literature.

6.1. Case Study 1: Abnormality Detection in a Smart Home

This use case pertains to the example depicted in Figure 6.1: the Bremen Ambient Assisted Living Lab has (automatic) sliding door. Sometimes a box or a chair accidentally blocks the door such that it opens only half way. In this case, the planning component in the overall system should be able to postdict such an abnormality and to find an alternative route for robotic vehicles which would usually pass the defect door.

A simplified domain description is as follows:

```
(:action open_door :effect if ¬ab_open then is_open)
(:action drive :effect if is_open then in_liv)
(:action sense_open :observe is_open)
(:init ¬is_open)
(:goal weak in_liv)
```

Listing 6.1: Simplified problem of moving through a door with potential abnormalities

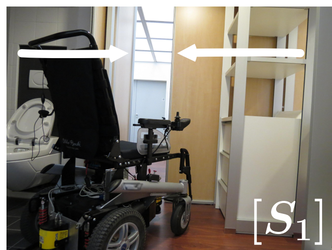
An action `open_door` causes a door to be open if there is no abnormality (denoted by `ab_open`).[1] The action `drive` has the effect that the robot is in the living room (which is behind the door) if the door is open. `sense_open` can be executed to determine the open-state of the door. Initially the door is not open and the goal is that the robot is in the living room.

6.1.1. Trace: Conditional Planning with Abnormalitiy Postdiction

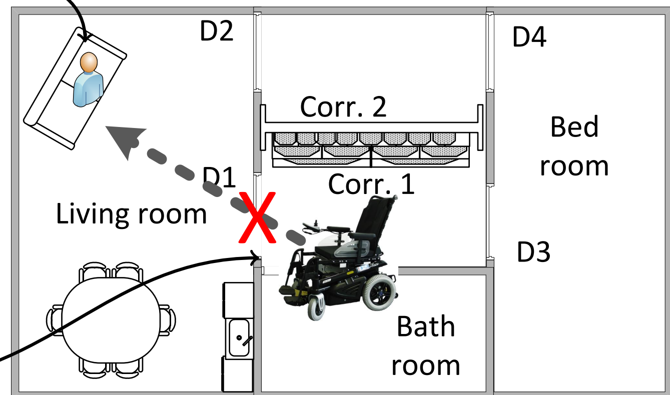
Consider the situation where a person instructs a command to reach a location, e.g. the sofa, to the wheelchair [S_0]. An optimal plan to achieve this goal is to pass D1. However, if D1 does not open because it is jammed, then a more error tolerant plan is required: [S_1] open D1 and verify if the action succeeded by sensing the door status. If the door is open, drive through the door and approach the user. Else there is an abnormality: in this case open and pass D3 [S_2]; drive through the bedroom; pass D4 and D2; and finally approach the sofa [S_3].[2] A transition tree is provided in Figure 6.2.
Initially, wheelchair *Rolland* is outside the living room ($\neg$`in_liv`) and the weak goal is that the robot is inside the living room. The robot can open the door (`open_door`) to the living room. Unfortunately, since the door may be jammed, opening the door does not always work, i.e. there may be an abnormality. However, the robot can perform sensing to verify whether the door is open (`sense_open`) and then postdict whether or not there is an abnormality in opening the door.
This mechanism is illustrated in Figure 6.2. Initially (at step $n = 0$ and branch $b = 0$) it is known that the robot is in the corridor at step $t = 0$ (denoted by $knows(\neg\texttt{in_liv}, 0, 0)$). The first action is opening the door, i.e. the Stable Model contains the atom $occ(\texttt{open_door}, 0, 0)$. Inertia holds for $\neg\texttt{in_liv}$, because nothing happened that could have initiated `in_liv`. Consequently, rules (F3a) – (F3c) trigger $kNotInit(\texttt{in_liv}, 0, 0, 0)$ and (F3f) triggers $knows(\neg\texttt{in_liv}, 0, 1, 0)$. In turn, the forward inertia rule (F3d) causes atom $knows(\neg\texttt{in_liv}, 1, 1, 0)$ to hold. Next, sensing

[1] The usage of abnormality-predicates for failure diagnosis is discussed in more detail in Section 7.2.
[2] Abnormalities are considered on the alternative route as well but skipped here for brevity.

[S_0]: Wheelchair is called using remote control or other input device.



[S_1]: Door is jammed.



[S_2]: Wheelchair takes alternative route.

[S_3]: Destination reached.

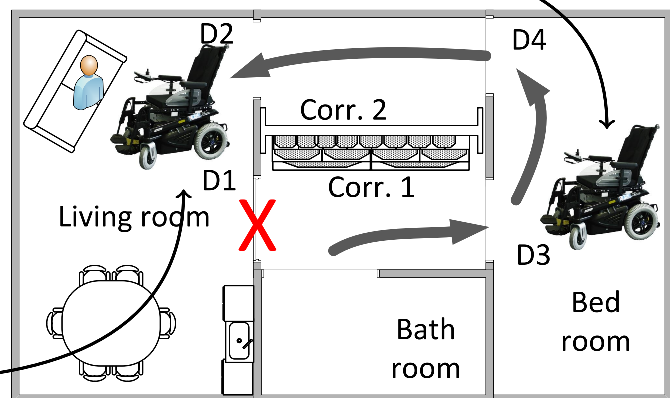


Figure 6.1.: Use case 1: abnormality detection in the Smart Home *BAALL*

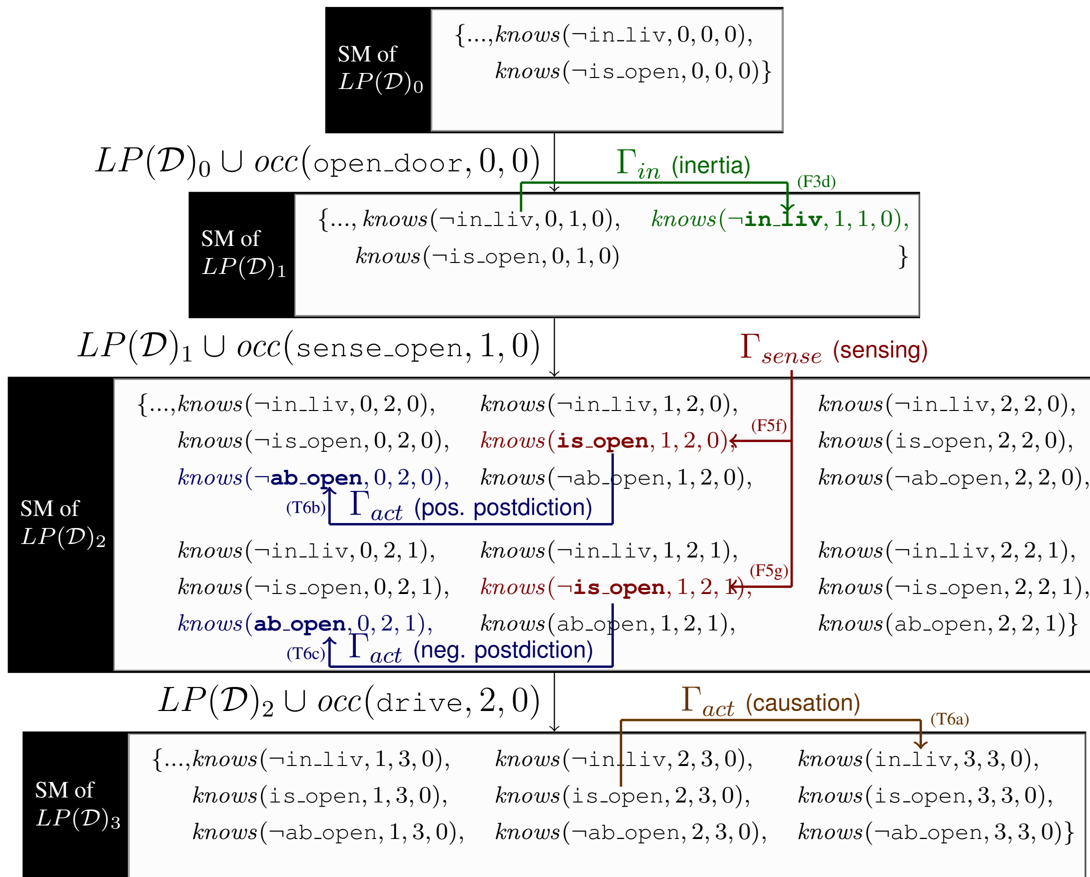


Figure 6.2.: Abnormality detection as postdiction with *h-approximation*

happens, i.e. $occ(\texttt{sense_open}, 1, 0)$. According to rule (F5f), the positive result is assigned to the original branch and $sRes(\texttt{is_open}, 1, 0, 0)$ is produced. With rule (F5g), the negative sensing result is assigned to some child branch $b' = 1$. In the example we have $sRes(\neg\texttt{is_open}, 1, 0, 1)$, such that together with (F5k) $knows(\neg\texttt{is_open}, 1, 2, 1)$ is produced. This result triggers the negative postdiction rule (T6c) and knowledge about an abnormality concerning the opening of the door is produced: $knows(\texttt{ab_open}, 0, 2, 1)$. Consequently, the wheelchair has to follow another route to achieve the goal.

For branch 0, we have $knows(\texttt{is_open}, 1, 2, 0)$ after the sensing. This result triggers the positive postdiction rule (T6b): because $knows(\neg\texttt{is_open}, 0, 2, 0)$ and $knows(\texttt{is_open}, 1, 2, 0)$ hold, one can postdict that there was no abnormality when $\texttt{open_door}$ occurred: $knows(\neg\texttt{ab_open}, 0, 2, 0)$. Finally, the robot can drive through the door: $occ(\texttt{drive}, 2, 0)$ and the causation rule (T6a) triggers knowledge that the robot is in the living room at step 3: $knows(\texttt{in_liv}, 3, 3, 0)$.

6.1.2. Results and Discussion

In the depicted scenario, the *weak* goal of driving to the couch was issued and the ASP solver found a plan within 140 ms on a standard 2Ghz Intel i5 computer with 6GB RAM. The purpose of this use case was primarily the illustration of the postdiction mechanism of the ASP implementation of $\mathcal{HPX}$. For a more thorough evaluation in terms of computation time we refer to the results obtained from the second scenario (Section 6.2.2) and the empirical comparison with other planners in Section 6.3.

6.2. Case Study: Interleaving Action Planning, Abnormality Detection and Abductive Explanation in a Smart Home

This case study emphasizes how planning, abnormality detection, abductive explanation and plan repair play together. The scenario takes place in the Bremen Ambient Assisted Living Lab (BAALL) and involves the autonomous robotic wheelchair "Rolland". We assume abnormalities in the wheelchair's driving action and illustrate how this is coped with in an online manner. In addition, an exogenous action happens which triggers abductive explanation and online plan repair. The Case Study is depicted in Figure 6.3. [3]

6.2.1. Trace: Interplay Between Controller and ASP Solver

The controller serves as interface between the robotic sensors and actuators and the ASP solver. It also translates goals which are received by human interface devices like remote controls, mobile phones or speech recognition systems into Logic Programming facts. For example, as illustrated in Figure 5.1, a person called `fred` is sitting on the couch and wants to get to the bathroom. He issues this goal using natural language and the controller of the online architecture employs its speech recognition module to generate an LP fact $LP(\mathcal{G}) =$ `wGoal(inroom(fred,bath))`. This fact is sent to the ASP solver which starts to compute Stable Models. The first Stable Model found (SM_1) is sent to the controller again which interprets it as a conditional plan. The plan foresees to use the wheelchair `rolland1` which is currently behind the couch to drive in front of the couch, pick `fred` up and bring him to the bathroom. During this course of actions, the wheelchair executes sensing actions to verify whether driving commands were successful.
To execute the plan, the controller translates the action occurrences in the Stable Model to corresponding XML strings which can be interpreted by the Smart Home and the

[3] A video of this use case can be found at `www.commonsenserobotics.org`, accessed Dec. 12th, 2013.

wheelchair. It also generates LP messages involving `exec/2` and `sensed/2` atoms to inform the ASP solver about action execution and sensing results.
First, `rolland1` receives the command to drive directly to the `couch`. However, unfortunately the passage is blocked by an obstacle (the little box) which was accidentally placed next to the couch. This abnormality is postdicted if the sensing action `senseloc(rolland1,couch)` reveals that the wheelchair is not at the couch (represented by the LP atom `sensed(¬inroom(rolland1,couch),1)`).
The ASP solver receives this information and according to the LP rule (FO5e) atom `sRes(inroom(rolland1,couch),1,0,0)` which contradicts the actual sensing result is no longer produced. Hence the assumptions which are required to achieve the goal are not met and the original plan SM_1 becomes invalid.
This triggers the ASP solver to generate new Stable Models. SM_2 represents a repaired plan which involves the second wheelchair `rolland2` to drive from the desk to the couch and to bring `fred` to the bathroom. The controller executes the proposed actions and while executing `drive_direct(rolland2,couch)` another person `george` walks into the bathroom and closes its door. The controller receives the sensing result that the door to the bathroom has been closed exogenously. This information is sent to the ASP solver in terms of a fact `sensed(closed(d5),4)` and the ASP solver explains the closing of the door with the occurrence of an exogenous event `exoHappens(person_close_door(george,d5),3,1)`. This is produced by the choice rule (FO9). The occurrence of the exogenous action triggers the postdiction that `george` must be in the bathroom. However, for privacy reasons the system is not allowed to open the bathroom door if the room is occupied, so the action `move_person(george,corr1)` must occur before the door can be opened.[4] Once door `d5` to the bathroom is opened, the wheelchair can drive into the bathroom and the goal is achieved.

6.2.2. Results and Discussion

To evaluate the practical performance of the $\mathcal{HPX}$ implementation we investigated the computation time required to solve the depicted use case. For the empirical evaluation of the scenario we use a slightly different formulation of the use case which can be found in Appendix C, Listing D.5.[5] Table 6.1 summarizes the results on a 2.3Ghz Intel i5 computer with 6GB RAM. We used the beta-version 3.0.92 of the online ASP solver *oclingo* for the experiments.
To investigate how the static relations which we describe in Section 5.2.4 improve the performance we present results for two cases. We implemented the scenario once with

[4]Since this action can not be controlled by the system, it is announced by the Smart Home's multimedia devices to emulate the execution. The success of the action is determined by user input, i.e. either George or Fred has to inform the system that the action has been executed. This can be done with a smart phone, via speech recognition or any other input device.

[5]The use case presented in this section was simplified for illustration purposes.

Figure 6.3.: Use case 2: interleaving planning, plan execution and abductive explanation

and once without the extended rules covering static relations and impossible actions (TO1), (TO2).[6] These prune the search space in that they guide the planner to consider only those actions which are actually possible. For example, consider the action of driving `rolland2` directly from the bedroom to the couch. The is not possible because there is no direct connection between bedroom and couch. Consequently the planner does not have to consider this action when generating the search space.

Reasoning task	**With static relations**	**Without static relations**
First plan found	4.36 sec.	14.58 sec.
1st Plan repair (abnormality)	23.65 sec.	50.89 sec.
2nd Plan repair (exogenous event)	4.8 sec.	7.71 sec.

Table 6.1.: Computation time required to solve online planning tasks

Table 6.1 depicts the computation times for the individual episodes of the use case:

1. *First plan found* denotes the time the ASP solver needed to find a first plan to bring `fred` to the bathroom. This refers to the naive plan of using `rolland1`, assuming that there is no abnormality.

2. *1st Plan repair (abnormality)* denotes the time the ASP solver needed to find an alternative plan when the sensing result that the wheelchair is not at the couch was received.

3. *2nd Plan repair (exogenous event)* denotes the time the ASP solver needed to find an alternative plan when the exogenous event of closing the bathroom door was abduced.

It is easy to see that the pruning mechanism, i.e. the consideration of static relations, approximately halves the computation time. To investigate how the individual subprocesses are involved in the Stable Model generation we enabled verbose output of the ASP solver. This told us that for the first case (with static relations) grounding played a minor role in the solving process, i.e. approximately 10% of the total computation time. For the second case (without static relations) grounding required 20-30% of the total computation time.
It shall also be noted that the time which the $\mathcal{HPX}$-compiler needs to translate the PDDL-like problem description into the Logic Programming rules was always in the order of milliseconds and was neglected.

[6]For the case without static relations we used the non-incremental translation rules (T1), (T2).

Problem	**h-appx.**	**ASCP**	**CFF**
RINGS(1)	0.016	0.016	0.01
RINGS(2)	0.203	0.156	0.02
RINGS(3)	**9.734**	169.26	0.09
RINGS(4)	MEM	MEM	1.56
RINGS(5)	MEM	MEM	480.1
SICK(2)	0.078	0.031	0.02
SICK(4)	0.889	0.061	0.04
SICK(6)	30.81	0.125	0.07
SICK(8)	1361.5	157.3	0.1
BTS(2)	0.016	0.016	0.01
BTS(4)	0.14	0.05	0.02
BTS(6)	7.160	0.55	0.03
BTS(8)	281.0	138.6	0.05

Table 6.2.: Comparison of different planners for benchmark problems from literature

6.3. Empirical Comparison with other Planners

In Section 6.2 we have investigated the computational performance of the online planning framework for a real-word use case. To evaluate how fast the $\mathcal{HPX}$-planning system performs wrt. other planners, we compare our planner with ContingentFF (CFF) by Hoffmann and Brafman (2005) and ASCP by Tu et al. (2007) using typical benchmark problem domains from the literature. The domains are selected on the basis of the individual features of the planners, namely postdiction, the performance optimization described in Section 5.2.4 and different sensing capabilities. A summary is presented in Table 6.2.

6.3.1. Benchmark Problems

Rings – RING(n_r)

A number (n_r) of rooms are ring-wise connected. There are windows in the rooms which must be closed and locked. It can be sensed whether the windows in the rooms are open or not (see e.g. (Cimatti et al., 2003)).

The connectedness of rooms is modeled with static relations and we have chosen this problem in order to investigate how static relations improve the planning performance.

Sickness **– SICK(n_s)**

A patient is infected with one of n_s illnesses. A test can be performed which stains a paper, such that the color of the paper indicates the illness. This domain requires postdiction to diagnose the illness based on the paper's color (see e.g. (Tu et al., 2007)).[7]
We have chosen this problem to investigate the influence of different sensing capabilities on the computation time: ASCP can sense the color of the paper with one action while $\mathcal{HPX}$ requires to perform n_s sensing actions: one for each possible color.

Bomb in the toilet **– BTS(n_p)**

In this problem, n_p is a parameter which reflects the number of suspicious packages that may or may not be bombs. Potential bombs can be disarmed by dunking them in a toilet. The agent can perform a `sense_metal`-action to sense whether or not a package contains a bomb and it can dunk a package into a toilet to disarm it if it contains a bomb (see e.g. (Weld et al., 1998)).
We have chose this problem because both ASCP and $\mathcal{HPX}$ do not have any advantage in the domain. Hence, this problem reflects an unbiased benchmark to compare ASCP and $\mathcal{HPX}$ if no postdiction is required.

6.3.2. Setup

All goals in the domains are *strong goals*. Experiments were conducted on a Intel i5 machine with 6 GB RAM. The ASP solver used is *clingo* (Gebser et al., 2011b). For the computation times of $\mathcal{HPX}$ we neglected the time it took the $\mathcal{HPX}$ compiler to translate the PDDL-like input syntax into Logic Programming rules; this was always in the order of magnitude of milliseconds.

6.3.3. Discussion

Even though CFF is based on a $\mathcal{PWS}$ and hence has a higher computational complexity than $\mathcal{HPX}$ and the 0-approximation-based ASCP planner, it clearly outperforms both approaches. This is probably due to heuristics that are based on relaxed formulations of the planning problem (see (Hoffmann and Brafman, 2005) for details).
The good result in the RING domain of $\mathcal{HPX}$ compared to ASCP is due to the fact that the connectedness of the rooms can be modeled as static relations, which gives our planner an advantage. However, the h-approximation is outperfomed by ASCP in the SICK domain because ASCP supports sensing the color of the paper with only one

[7] To "emulate" postdiction for ASCP, the definition of additional static causal laws was necessary, similar to the case presented in Example 2.1.

sensing action while $\mathcal{HPX}$ has to consider $n_s - 1$ sensing actions: one for every potential illness. That is, $\mathcal{HPX}$ requires a higher planning horizon.
The results of these experiments, in particular the clear superiority of CFF, leads to the conclusion that implementing heuristics in terms of Answer Set Programming is a promising future research direction to keep ASP based planners competitive with dedicated and more optimized planners implemented in traditional programming languages like C++. We discuss this point in Section 7.2.
A problem for benchmarking planning with incomplete knowledge is that only a few implemented planning systems which are able to cope with incomplete knowledge and sensing actions are freely available. Table 2.3 enlists many epistemic action theories, but it also shows that only few of them are actually implemented. From these theories which are actually implemented, not all are freely available and others have a different understanding of "planning": while in this work planning is understood as finding a course of actions that lead from an epistemic initial state to an epistemic goal state (see Definition 2.4), other systems like INDIGOLOG system (de Giacomo and Levesque, 1998) understand planning as executing a predefined course of actions, possibly enhanced with dynamically generated sub-plans. Though in INDIGOLOG, the dynamic generation of sub-plans is equivalent to our definition of planning, INDIGOLOG is not capable of dynamically integrating *sensing actions* in a plan. Another example is the EFEC implementation by Miller et al. (2013): even though the implementation exists and is available online, their planning mechanism does not feature a semantics based on *branching*. These differences make it hard to implement planning problems for different planners in a way such that their formalizations and results are actually comparable.
Another problem is that of syntactic compatibility. The the Planing Domain Definition Language PDDL (McDermott et al., 1998) – which is the de-facto standard language to specify planning problems – does not officially support sensing actions and incomplete knowledge. Hence, each planner uses its own input language or dialect.

7 Discussion and Future Work

To conclude this thesis, we discuss its scientific contribution in Section 7.1. A particular emphasis is given on the interplay of the features of $\mathcal{HPX}$ and on its practical application in robotics and related fields. We also identify limitations of $\mathcal{HPX}$. This connects to Section 7.2 where we sketch how limitations could be overcome and where we propose future extensions for $\mathcal{HPX}$. Finally, Section 7.3 provides an upshot of the overall achievements of this thesis.

7.1. Discussion

The research question which we state in the introduction of this thesis is formulated as follows:

How is it possible to realize temporal postdictive reasoning *whilst avoiding a combinatorial explosion of state variables?*

As a main contribution we have presented the *h-approximation* ($\mathcal{HPX}$) of knowledge which answers this question: it avoids the combinatorial explosion of state variables by approximating the agent's knowledge state, but it is still capable of postdiction because the temporal dimension of knowledge is explicitly represented. The temporal dimension makes it possible to efficiently postdict about facts. At the same time it makes $\mathcal{HPX}$ more expressive than most other epistemic action theories which do not consider temporal knowledge.

$\mathcal{HPX}$ has a particular combination of features which in Table 2.3 was compared with other epistemic action theories. In the following we discuss how the interplay of these features creates a synergistic gain in practical applications. Specifically, we discuss the advantages which emerge if combining the complexity properties of $\mathcal{HPX}$ with the support for postdictive reasoning; in addition we demonstrate that the temporal

knowledge dimension of $\mathcal{HPX}$ is useful for planning problems which involve concurrent acting and sensing.

Postdiction with a Linear Number of State Variables

The core feature of $\mathcal{HPX}$ is its support for postdictive reasoning at a comparably low computational complexity. The low complexity emerges from the fact that only a linear number of state variables are required to model an agent's knowledge state, as compared to an exponential number for existing theories. The combination of both features is particularly useful in practical applications: *(a)* a low computational complexity lays the ground for the application in practical applications where real-time response to planning queries and other reasoning tasks is needed. Algorithms with a higher complexity generate a combinatorial explosion of state variables and can only be used in very small problem domains, otherwise computation time becomes unacceptable. *(b)* Postdiction can be used to achieve error-tolerance: in practice, actions do not always succeed due to unforeseen complications and system failures. In Chapter 6 we have demonstrated that a way to diagnose such abnormalities is postdiction.
Another advantage before $\mathcal{PWS}$-based approaches becomes obvious when considering that in $\mathcal{PWS}$-based action theories the number of state variables is exponential wrt. the number of *unknown* fluents. In contrast, with $\mathcal{HPX}$ the number of state variables is *independent* from the number of unknown fluents. A synergistic gain emerges if the independence of the number of state variables from unknown fluents is combined with postdictive reasoning for abnormality detection: under the (realistic) assumption that the outcome of actions is always subject to potential failure one has to model abnormalities as unknown conditions of actions. These can be postdicted by observing whether the action was successful or not. However, the more actions a domain contains, the more abnormalities have to be modeled and hence the more unknown fluents are involved. This makes the application of $\mathcal{PWS}$-based theories inappropriate due to the exponential blowup of possible worlds caused by unknown abnormalities. $\mathcal{HPX}$ will perform much better in such real-world cases because the exponential blowup is avoided.

Temporal Dimension of Knowledge, Postdiction and Concurrent Acting and Sensing

The $\mathcal{HPX}$ formalism is more expressive than most epistemic action theories in the sense that it supports reasoning about the past. As we demonstrate with Example 7.1, this temporal aspect combines nicely with postdiction and concurrent acting and sensing. Example 7.1 is an extended version of the well-known Yale Shooting Problem (Hanks and McDermott, 1987). It shows that the temporal dimension of knowledge is required to reason about actions which sense a fluent's value and concurrently change this value:

pulling the trigger of a gun causes one to sense (by hearing the explosion) whether the gun was loaded. At the same time the gun unloads. If the gun was loaded then one can conclude that the target is dead. This inference is not possible with existing theories.

Example 7.1 An Extended Yale Shooting Problem
Consider the following domain specification:

```
(:init alive)
(:action shoot
    :effect ¬loaded
    :effect if loaded then ¬alive
    :observe loaded)
```

The turkey is initially alive, but it is unknown whether the gun is loaded. Shooting unloads the gun and causes the turkey to be dead if it is loaded. In addition, shooting causes to know whether the gun was loaded. The task is to infer whether the turkey is dead after the shooting, depending whether or not the firing of the gun was perceived. In $\mathcal{HPX}$ this is possible, because sensing yields the value of a fluent *before* the action takes place. That is, if the action is executed at $t = 0$ then in the resulting state $t = 1$, knowledge about the loaded-ness at $t = 0$ is produced, even if this differs from the `loaded`-ness in the resulting state $t = 1$. According to Definition 3.2 about initial knowledge we have

$$\mathfrak{h}_0 = \langle \{\}, \{\langle \texttt{alive}, 0\rangle\}\rangle$$

The $\mathcal{HPX}$ transition function (3.7) evaluates as:

$$\begin{aligned}\Psi(\texttt{shoot}, \mathfrak{h}_0) =& \{\mathfrak{h}_1^+, \mathfrak{h}_1^-\}, \text{ where}\\ \mathfrak{h}_1^+ =& eval(\langle\langle \texttt{shoot}, 0\rangle, \{\langle \texttt{alive}, 0\rangle\} \cup \{\langle \texttt{loaded}, 0\rangle\}\rangle\\ \mathfrak{h}_1^- =& eval(\langle\langle \texttt{shoot}, 0\rangle, \{\langle \texttt{alive}, 0\rangle\} \cup \{\langle \neg\texttt{loaded}, 0\rangle\}\rangle\end{aligned}$$

The $eval$ function calls $cause$ (3.13), and in the case of $\mathfrak{h}_1^+$ this correctly generates knowledge that the turkey is dead after shooting, i.e. $\mathfrak{h}_1^+ \models \langle \neg\texttt{alive}, 1\rangle$. In the case of $\mathfrak{h}_1^-$ $cause$ correctly generates knowledge that the turkey is still `alive` after shooting: $\mathfrak{h}_1^- \models \langle \texttt{alive}, 1\rangle$.

The example illustrates that the temporal dimension of knowledge is required in scenarios where the temporal details of sensing and physical action effects play a role. More examples can be found in areas like narrative interpretation or forensic reasoning, where information about the past is gained e.g. through the statement of a witness.

Incompleteness of $\mathcal{HPX}$

In Section 3.3 we show that the computational complexity of solving the plan-existence problem is in NP, and hence one level below the Σ_2^P complexity of the same problem in the $\mathcal{PWS}$-based $\mathcal{A}_k$semantics. The price to pay for this is that $\mathcal{HPX}$ is not complete wrt. $\mathcal{PWS}$-based approaches. Even though $\mathcal{HPX}$'s postdiction mechanism allows it to solve more problems than other approximate theories which are also in NP, there are still problems where no solution can be found. Intuitively, these are problems where knowledge is generated because the same fluent in different possible worlds obtains the same value. Consider Example 7.2 as a minimal example where knowledge is generated with a $\mathcal{PWS}$-based approach but not with $\mathcal{HPX}$. Example 7.2 shows that there are cases where $\mathcal{HPX}$ is incomplete. However, it also shows that in many cases it is possible to work around this issue with an alternative domain modeling.

Implementation as ASP and its Limitations

Answer Set Programming is a general approach to solve NP-complete search problems, like the planning problem with the h-approximation. ASP solvers like clingo (Gebser et al., 2012b) employ highly efficient algorithms and can act as workhorse to solve planning problems without the need to implement a planner in a traditional programming language like C++.
Also, using ASP as reasoning engine for planners is a relatively well-understood method. In particular, the Negation as Failure (NaF) semantics of ASP is a convenient alternative to circumscription for realizing the non-monotonicity of action theory (see Section 2.1.3). Finally, the use of ASP allows us to formally prove that the results obtained by the solver are actually sound wrt. the underlying $\mathcal{HPX}$ theory. Such a proof is very circumstantial for planners which are implemented in traditional programming languages like C++ because those programming language typically have a much more complex semantics.
However, we identify two limitations of the ASP formalization which both are caused by the fact that ASP uses no quantification. Lee and Palla (2009) show that there are ways to express quantification in Answer Set Programs using an extended First-Order ASP semantics, but this only works for certain canonical cases. So far we did not find a possibility to solve the following problems:

- In the operational semantics we are able to express that $inertial(l, t, \mathfrak{h})$ holds if there exists no condition literal in an effect proposition which would cause $\bar{l}$ to hold (see 3.10). We found no simple solution to capture the $\exists$-quantification over condition literals with Answer Set Programming. Instead we use the $kNotSet(\bar{l}, t, b)$ predicate to represent that a literal l is inertial and implement rules (F3a) – (F3c) to capture a similar behavior as in the operational semantics. However, this way of implementing inertia is only correct in combination with rule (2) which forbids

Example 7.2 Incompleteness of $\mathcal{HPX}$
Consider the following action which sets a fluent f to $\neg f$ if f is true.

```
(:action falsify(f) :effect if f then ¬f)
```

Assume that initially it is unknown whether f of $\neg f$ holds and compare how the execution of this action affects knowledge in (1) a $\mathcal{PWS}$-based approach and (2) in $\mathcal{HPX}$:

1. If f is initially unknown, then in a $\mathcal{PWS}$-based approach the agent's knowledge state is represented by two possible worlds which we denote by $\Sigma_0 = \{\{f\}, \{\neg f\}\}$. If action `falsify`$(f)$ is applied to Σ_0, then its effect proposition `if` f `then` $\neg f$ is applied to both possible worlds resulting in a successor state Σ_1. Informally:

   $$\Sigma_0 = \{\{f\}, \{\neg f\}\} \xrightarrow{\texttt{falsify}(f)} \Sigma_1 = \{\{\neg f\}, \{\neg f\}\}$$

   That is, the first possible world $\{f\}$ in state Σ_0 becomes $\{\neg f\}$ in the successor state Σ_1. Since a fluent literal is known to hold if it is true in all possible worlds, a $\mathcal{PWS}$-based semantics correctly represents that $\neg f$ is known to hold after executing `falsify`(f).

2. With the h-approximation, the initial state would be $\mathfrak{h}_0 = \langle \emptyset, \emptyset \rangle$. Applying `falsify`$(f)$ evaluates as follows:

   $$\Psi(\texttt{falsify}(f), \mathfrak{h}_0) = \{\langle \langle \texttt{falsify}(f), 0 \rangle, \emptyset \rangle\}$$

   That is, the agent does not acquire any new information about f after executing `falsify`(f). The only way to generated knowledge in $\mathcal{HPX}$ is either through sensing or one of the inference mechanisms **IM.1 – IM.5** of which none applies in this case.

To see how one can work around the incompleteness problem consider the following non-conditional action `falsify2`:

```
(:action falsify2(f) :effect ¬f)
```

In practice the outcome of `falsify` and `falsify2` is identical in that $\neg f$ will always hold after execution. $\mathcal{HPX}$ correctly generates knowledge if `falsify2` is used instead of `falsify`.

that two effect propositions (EPs) with the same effect literal can not be applied simultaneously.

- In the operational semantics, *positive postdiction* is realized with the function $add_{pd^{pos}}(\mathfrak{h})$ which involves a $\forall$-quantification over effect propositions. There is no simple way of modeling this in terms of ASP, and therefore the ASP formalization of positive postdiction (rule T6b) also relies on restriction (2) (two EPs with the same effect literal can not be applied simultaneously).

The restriction that two similar effect propositions can not be applied simultaneously is not necessary in the operational semantics. Therefore the ASP implementation will not generate solutions in cases where two or more similar EPs are applied simultaneously, even though the operational semantics does.

Semantics of Online Planning and Abductive Explanation

Based on the offline ASP formalization we present extensions which make $\mathcal{HPX}$ capable of performing incremental online planning with abductive explanation.
However, the semantics of the execution monitoring mechanism itself is not modeled in our theory. This would require one to model the system architecture depicted in Figure 5.1 in terms of an operational or model-theoretic semantics on top of the original $\mathcal{HPX}$ semantics. This is a research endeavor on its own and out of scope for this thesis. However, due to its inherent postdiction capabilities $\mathcal{HPX}$ could serve as a basis for an execution monitoring semantics.
Another point for discussion concerns the abductive explanation extension presented in Section 5.2. A problem is that there may be multiple explanations for unexpected world property changes. For example, in the second use case (Section 6.2) the closing of a door was explained by the occurrence of an exogenous action: an external agent, the person "George", closed the door. However, the door could also have been closed by an air breeze and without additional external knowledge it is impossible to determine which explanation is true.
If the explanation is wrong, and if the action which is used in the explanation has a conditional effect then this causes additional problems because false knowledge about the conditions of the action could be postdicted. In the scenario from Section 6.2 for example, a conditional effect is that George can open a door if he is in a room adjacent to the door. For the bathroom door this is either the corridor or the bathroom. Consider that the closing of the door is detected, and the explanation that George closed the door from the corridor has been chosen. Then the system will postdict that George is in the corridor since this is a condition which must hold for George to open the door. However, if in reality George closed the door from the bathroom then this postdiction is wrong.

A partial solution is to restrict exogenous actions to have only one effect literal and no conditions. In that case, even though explanations about the occurrence of actions may be wrong they do not have side-effects on knowledge.

7.2. Future Work

Though $\mathcal{HPX}$ is already capable of solving many problems in practical applications there are many possible improvements in both theory and application of $\mathcal{HPX}$.

An Elaborate Temporal Semantics for $\mathcal{A}_k$

In Section 3.4 we define $\mathcal{A}_k^{TQS}$, the *temporal query semantics* for the action language $\mathcal{A}_k$. This extended semantics serves the purpose to provide a formal semantic grounding and to define soundness of $\mathcal{HPX}$. However, its current definition is limited in that it only considers sequences of actions. Conditional plans and concurrency are not supported so far. A generalization to address non-boolean fluent values is also useful. Elaborating $\mathcal{A}_k^{TQS}$ with these extensions could result in a theory which serves as a benchmark for other temporal epistemic formalisms such as EFEC (Miller et al., 2013),[1] in a similar way in which $\mathcal{A}_k$ became a benchmark for non-temporal epistemic action formalisms.

Ramifications and Static Causal Laws

Ramifications concern the side-effects of actions. For example, if a robot carries an object and the robot moves, then the object moves with the robot. This side-effect can in principle be modeled in the current version of $\mathcal{HPX}$, but in order to achieve this, the action specifications have to be modeled in a very circumstantial and elaboration intolerant manner. A solution to this so-called *ramification problem* is using Static Causal Laws (SCL) (see e.g. (Turner, 1999)). SCL are constructs of the form *if* $l_1, \ldots, l_n$ *then* l_{scl} which state that if literals $l_1, \ldots, l_n$ are initiated, l_{scl} is also initiated.[2]
One way to include SCL in $\mathcal{HPX}$ is to compile them into the effect propositions of actions. For instance, the following action represents a simple `move`-action:

```
(:action move
  :parameters (?r - Robot ?from ?to - Location)
  :effect (and !at(?r, ?from) at(?r, ?to)))
```

[1] In a personal conversation with Rob Miller he mentioned that a temporal epistemic framework like $\mathcal{A}_k^{TQS}$ would be useful to define soundness of temporal theories like EFEC (Miller et al., 2013). This underpins our observation that $\mathcal{A}_k^{TQS}$ can play an important benchmark role in the area of epistemic action theory.

[2] For a detailed description of the semantics of SCL we refer to (Turner, 1999).

A SCL which represents that the object held by the robot is always at the robot's location could be written in a PDDL-style as:[3]

```
(:scl-holdObject
  :parameters (?r - Robot ?o - Object ?loc - Location)
  :scl (and holding(?r,?o) at(?r, ?loc)) -> at (?o,loc))
```

Integrating the SCL into the above action would create an additional action:

```
(:action move-holdObject
 :parameters (?r - Robot ?from ?to - Location ?o - Object)
 :effect (and !at(?r, ?from)
         at(?r, ?to)
         (if holding(?r,?o) then (and !at(?o,?from) at(?o,?to)))))
```

An extension to $\mathcal{HPX}$ would mean to automatize the generation of additional action definitions such that SCL are considered. Tu et al. (2007) have shown how to integrate SCL within the 0-approximation semantics of $\mathcal{A}_k$, but it is unclear how this automation is to be be realized wrt. $\mathcal{HPX}$'s postdiction rules and the temporal dimension of knowledge.

Resources, Functional Fluents, Quantities and Simple Arithmetics

Resources are quantities, like e.g. the remaining power of a battery. In the presented $\mathcal{HPX}$ framework, quantities have to be modeled in a circumstantial way. Values have to be discretized and this causes a huge number of objects in the domain specification. Epistemic planning systems like MAPL (Brenner and Nebel, 2009) or PKS (Petrick and Bacchus, 2004) do not have this limitation and show how this problem can be solved. A first step towards modeling quantities and resources is to introduce *functional fluents*. $\mathcal{HPX}$ only allows one to model binary fluents, and currently functional fluents have to be emulated with binary fluents as follows: let the number of energy states be discretized into 100 values, and a binary predicate `hasEnergy` represents the robot's energy level. Then to state that the robot's current energy level is e.g. 78 one has to write `hasEnergy(robot, 78)`, `¬hasEnergy(robot, 1)` ..., `¬hasEnergy(robot, 77)`, ..., `¬hasEnergy(robot, 79)`, ..., `¬hasEnergy(robot, 100)`. With a functional fluent semantics the same could be written with a single statement: `hasEnergy(?r) = 78`.
A second step is to combine functional fluents with simple arithmetics. This allows one to natively model basic mathematical operations with quantities, such as the usage of energy. As an example consider how the following energy-consuming move action is modeled:

[3] There is a semantics difference between *axioms* in PDDL 1.0 (McDermott et al., 1998) and SCL as described e.g. in (Tu et al., 2007): SCL are always "triggered" by an action, while *axioms* are laws that hold universally. For details consider (McCain and Turner, 1995).

```
(:action move-consume
  :parameters (?r - Robot ?from ?to - Location)
  :vars ?startE = hasEnergy(?r)
        ?consumption = consumes(?r, ?form, ?to)
  :effect (and  !at(?r, ?from) at(?r, ?to)
                !hasEnergy(?r,?startE)
                hasEnergy(?r,?startE-?consumption)))
```

The syntax is similar to that of MAPL (Brenner and Nebel, 2009). the `:vars` section represents variable assignment. Instead of having `?startE` in the parameters of the action, `?startE` is obtained by evaluating the functional fluent `hasEnergy(?r)`. Hence, the number of instantiations of the action operator does not grow with the number of discretized energy values any more. Similarly, `?consumption` is a variable which represents the amount of energy that the robot requires for the particular instantiation of the drive action. In the effect specification there is a arithmetic "-" operation to compute the remaining energy.
A future research aspect is to implement and to define a functional fluent semantics with arithmetics, which would make it possible to model action operators like the energy-consuming move action. Of particular interest is the epistemic aspect of resources which involves postdiction of functional fluent values.

Deadlock Detection

The weak planning approach which we pursued in the Smart Home scenario (Section 6.2) has the practical advantage of a fast system reaction time, i.e. actions can already be executed even if every possible path to the goal has not been computed yet. The disadvantage of weak planning is that an agent might run into a deadlock. For instance, consider an agent which has the task to find an object in a building. Assume that it can execute a sensing action to determine whether the object is in the same room as the agent. Consider further that there are doors connecting the rooms, which are automatically closing after the agent moves through them, but there are some doors that can only be opened from one side. Then, if the agent passes such a door, the way back is blocked.
Finding a method to efficiently analyze a domain to find and avoid deadlocks is an important research question, for instance for robotic rescue and exploration tasks in unknown environments.

"True" Concurrency and Durative Actions

$\mathcal{HPX}$ is capable to model concurrent action execution, but this is only possible under the assumption that actions have the same duration. This causes a "patchy" system behavior in practice: consider the driving of a robot R1 and the opening of a door D. Opening the

door will usually take about 3 seconds, while the driving of R1 can take much longer. If the open-ness of the door is the condition for another action, e.g. for a second robot R2 which is about to move through that door, then R2 would have to wait until the first robot finishes its driving, even though the door may already be open. The reason for this is that the state transition is only complete if both concurrent actions are finished. A solution to this is the consideration of the duration of actions, which would allow one to realize "true" concurrency. However, it is unclear how this affects the complexity of the planning problem, and whether computation times remain acceptable for practical applications if considering time.

Performance Optimizations

The planning problem for $\mathcal{HPX}$ is in a lower complexity class than for $\mathcal{PWS}$-based approaches. However, since the problem is still in NP it is commonly considered to be intractable. The use cases presented in Chapter 6 have shown that even though the $\mathcal{HPX}$ planner is capable of performing assistance tasks in real-world environments, its computation time (e.g. 23.65 sec. for plan repair) is not acceptable for a seamless integration in a Smart Home or other real-world applications where a quick response to planning queries is required.
However, the highly optimized Contingent Planner CFF (Hoffmann and Brafman, 2005) for example shows that despite its even higher complexity computation times are in many cases tolerable (see Table 6.2).
This leads to the hypothesis that a planner based on the $\mathcal{HPX}$ theory can be even faster than CFF or other $\mathcal{PWS}$-based approaches, due to the lower computational complexity. We make three propositions to achieve this:

1. *Optimize the ASP solving parameters and order of LP rules.*
   ASP is fully declarative in the sense that the order in which the LP rules are stated does not affect the solutions. However, it is well known that the order in which the rules of an LP are stated can have an effect on the computational performance (see e.g. (Gebser et al., 2012b)). A way to optimize a Logic Program is to find a rule ordering which is more efficient in practice.

   In addition to the optimization of the order of LP rules, the solving process can also be optimized by adjusting the parameters of the LP solver properly. Solvers like *clingo* (Gebser et al., 2012b) provide different heuristics and so-called "nogood-learning" mechanisms and other parameters which can be adjusted such that they are more appropriate for the individual LP to solve. So far we have only tried the standard settings.

2. *Perform planning during plan execution.*
   For the online setup described in Section 6.2 we interleaved planning and plan

execution such that (1) a plan is generated, (2) the plan is executed until a sensing result revealed the plan to be invalid or until the goal is achieved, (3) if the goal is not achieved then the plan is repaired, (4) the repaired plan is executed until the plan becomes invalid or the goal is achieved, (5) the plan is repaired, etc.. Since in this interleaving execution takes a considerable amount of time, the plan extension can be shifted to take place during plan execution. That is, the solver could already extend its search space and pre-compute other plans which might solve the problem while the robot is executing the previous plan. This would lead to faster plan repair phases because parts of the search space can already be precomputed during the action execution phase.

3. *Implement heuristics in terms of ASP.*
   CFF and other PDDL-based planners like MBP (Cimatti et al., 2003) are very performant because they heavily rely on heuristics related to the specifics of action planning. Since ASP solvers are much more general than dedicated action planners, their heuristics have to be be more general as well. This means that they can not exploit certain specifities which only occur in action planning. We believe that identifying such specifities and encoding them as heuristics in the Logic Program is a promising way to improve the computational performance of ASP based planners in general.

   Heuristics which depend on the individual planning problem can also help to improve performance. For instance, one could reduce the search space of a navigation problem by not allowing that a door is opened and then immediately closed again. One can be even more restrictive by stating that whenever a door is opened, then the agent must move through the door. Similar approaches can be found in literature (e.g. the *control constructs* employed in the TALplanner by Kvarnström (2005)) and it was shown that this can drastically reduce the computation time.

Elaboration Tolerant Abnormality Detection – A Partial Solution to the Qualification Problem

The *Qualification Problem* (McCarthy and Hayes, 1969) is the problem of considering all conditions and qualifications under which an action has a certain effect. For instance, the driving of a robot is only successful if there is no obstacle blocking the way, but in an open world one can not model every possible obstacle for a drive-action.
A well-known partial solution to this problem is to consider abnormalities e.g. (Kvarnström, 2005; Patkos, 2010). In this work, we proposed to model an abnormality fluent for each action which represents whether the action will succeed. For example, the use case in Section 6.1 involves an action `open_door` which is only successful if an abnormality

fluent `ab_open` is false. At the current state of $\mathcal{HPX}$ this requires the domain-designer to manually specify abnormality fluents and respective effect propositions of actions. On the one hand, this is sensible because the domain-designer can himself decide which actions are unreliable, even on the level of the individual effect propositions. On the other hand, this method is not elaboration tolerant, because the domain-designer has to model abnormalities himself. A future research question is how to semi-automatize the integration of abnormalities in action effects such the domain designer can control the abnormality modeling to an appropriate extend.
A related issue refers to *dynamic abnormalities*. For instance, there may be an obstacle blocking the drive-action of a robot, but if this obstacle is a moving object (e.g. the family dog) then it might move after some time and the abnormality does not exist anymore. In general, if an action is unsuccessful then it may make sense to try again some time later. A solution to this problem can be to model abnormalities with a "decay", i.e. knowledge about abnormalities ceases after a certain time.

Integration in a General Cognitive Robotics Framework

So far, $\mathcal{HPX}$ is a stand-alone online planner with support for abductive explanation. An interface to an established Robotic Frameworks like ROS[4] would allow one to seamlessly use $\mathcal{HPX}$ as a reasoning tool in a huge number of robotic applications. For this reason, $\mathcal{HPX}$ is currently being integrated in the *ExpCog* (Suchan and Bhatt, 2013) Cognitive Robotics framework.
ExpCog is aimed at integrating logic-based and cognitively-driven agent-control approaches, qualitative models of space and the ability to apply these in the form of planning, explanation and simulation in a wide-range of robotic-control platforms and simulation environments. In addition to its primary experimental function, ExpCog is also geared toward educational purposes. ExpCog provides an easy to use toolkit to integrate qualitative spatial knowledge with formalisms to reason about actions, events, and their effects in order to perform planning and explanation tasks with arbitrary robot platforms and simulators. As demonstrators, support has been included for systems including *ROS*, *Gazebo*, *iCub*. The core integrated agent-control approaches include logic-based approaches like *Situation Calculus*, *Fluent Calculus*, or *STRIPS*, as well as cognitively-driven approaches like *Belief-Desire-Intention*. Furthermore, additional robot platforms and control approaches may be seamlessly integrated.

Explore Other Applications

This thesis focuses on the application of $\mathcal{HPX}$ in Cognitive Robotics. However, there are several other domains where a temporal epistemic theory is useful. An example is

[4]http://www.ros.org, accessed on 30th July 2013

narrative interpretation and forensic reasoning. For Instance, consider a criminal case where a witness states observations she made in the past. To reason abut this information one needs a theory like $\mathcal{HPX}$ which explicitly models time. Existing action theories are not capable of performing such temporal reasoning.

7.3. Summary

Epistemic action formalisms provide the theoretical backbone for deliberation tasks in Cognitive Robotics and related applications. However, existing formalism are either based on a possible-world-semantics, therefore suffering from a combinatorial explosion of state variables or they are approximations, incapable of performing postdictive reasoning.

This thesis fills this gap and shows that it is possible to implement postdictive reasoning without the need for an exponential number of state variables. The key is a temporal dimension of knowledge, which has the interesting side-effect of making the theory more expressive.

We only identified two approaches (Miller et al., 2013; Vlaeminck et al., 2012) which have a comparable temporal expressiveness. However, both approaches do not have a semantic grounding and soundness or completeness results wrt. other epistemic action theories.

In addition to the theoretical results, the thesis demonstrates the theory's applicability in practice by presenting its implementation and integration in a robotic framework.

A

Soundness of ASP Implementation wrt. $\mathcal{HPX}$ Semantics

This appendix contains results for the soundness relations between the ASP implementation of $\mathcal{HPX}$ and its operational semantics (see Table 4.1).
Section A.1 provides notational conventions and in Section A.2 we depict the general proof structure.
Section A.3 contains the main soundness proof for state transitions. It shows that if knowledge about a pair $\langle l,t\rangle$ (denoted by $knows(l,t,n+1,b)$ atoms) is generated by the occurrence of actions in the ASP implementation then knowledge is also generated by the $\mathcal{HPX}$-transition function, i.e. $\exists\mathfrak{h}' \in \Psi(\boldsymbol{A},\mathfrak{h}) : \mathfrak{h}' \models \langle l,t\rangle$. This relies on several other Lemmata which have a similar form and which are also stated and proven throughout the appendix: Section A.4 proves soundness of the application of effect propositions, Section A.5 proves soundness of sensing results and Section A.6 proves auxiliary Lemmata.

A.1. Notational Conventions

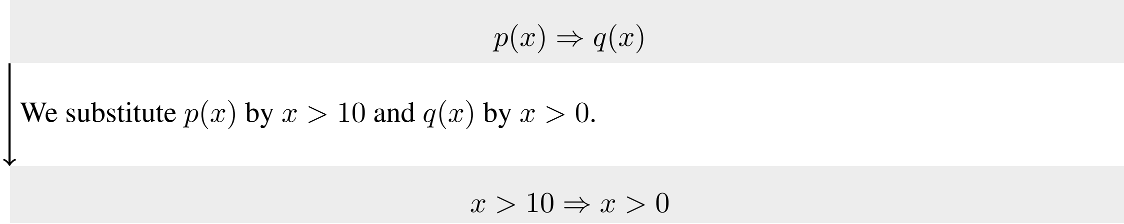
We presume the following notational conventions.

- `maxS` and `maxB` are constants denoting the maximal plan depth and width respectively. $0 \leq n \leq$ `maxS`, $0 \leq b \leq$ `maxB` and $0 \leq b' \leq$ `maxB` denote variables for steps and branches respectively.
- $\mathcal{D}$ is a domain description with the initial h-state $\mathfrak{h}_0$
- $LP(\mathcal{D})$ is the Logic Program of a domain description $\mathcal{D}$ without the plan-generation rule (F7) and without the goal statements generated by translation rule (T8)
- $S_{\mathcal{D}}^{\boldsymbol{P}}$ is a Stable Model of $LP(\mathcal{D}) \cup \boldsymbol{P}$ where $\boldsymbol{P}$ is a set of $occ(a,n,b)$ atoms with $0 \leq n <$ `maxS` such that

- $\forall a, n, b : \big(occ(a,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow uBr(n,b)\big)$.[1]
- $\forall n, b : \big(uBr(n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow \exists a : occ(a,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}\big)$ [2]

- $\boldsymbol{A}_{n,b} = \{a | occ(a,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}\}$ is a set of actions applied at a transition tree node with the "coordinates" $\langle n, b\rangle$.

The main proofs in this appendix concern implications which we mark by writing them in a gray box. Below the gray box we state and justify substitutions and generalizations we make and then we state the resulting implication in the next gray box. For example:



A.2. Proof Overview and Structure

The core soundness theorem to be proven is Theorem 4.1 which states that the model-theoretic interpretation of *state transitions* is sound wrt. the actual state transitions defined in the operational semantics.
The proof of this theorem involves several Lemmata which are implications of the following form:

$$\begin{aligned}\forall n, b, b' : \quad & hasChild(n, b, b', S) \Rightarrow \\ & \exists \mathfrak{h} \in \Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S)) : \\ & \qquad \forall x \in \mathcal{X} : \big(q(x, n+1, b') \in S \Rightarrow p_{op}(x, \mathfrak{h})\big)\end{aligned} \tag{A.1}$$

where $\mathcal{X}$ is a finite set of symbols, $q(x, n+1, b')$ denote atoms in the Stable Model S of a Logic Program and $p_{op}(x, \mathfrak{h})$ is a relation between x and the h-state $\mathfrak{h}$ (typically some form of entailment).
For example, for Lemma (A.2) $\mathcal{X}$ is the set of all pairs of literals and time steps. In that case, $q(x, n+1, b')$ corresponds to $knows(l, t, n+1, b')$ and $p_{op}(x, \mathfrak{h})$ corresponds to $\mathfrak{h} \models \langle l, t\rangle$.

[1] This restriction reflects the mechanics of the plan generation rule (F7) which only generates $occ(a,n,b)$ atoms if $uBr(n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}$.
[2] This restricts that there are no "gaps" in a plan, i.e. for all nodes in used branches there occurs at least one action. Note that this restriction is met for all $occ/3$ atoms which are generated by the plan generation rule (F7).

To prove implications of the form (A.1) we first eliminate the $hasChild(n, b, b', S)$ premise by considering two different cases where $hasChild(n, b, b', S)$ becomes true. This case distinction also helps to eliminate the $\exists$ quantification over h-states.
What remains are simple implications of the following form for each case.

$$\forall x \in \mathcal{X} : \big(q(x, n+1, b') \in S \Rightarrow p_{op}(x, \mathfrak{h})\big) \tag{A.2}$$

We prove the implications by complete induction over x, where X is a finite set. The induction consists of one or more base steps and induction steps. In the following we will demonstrate this induction. Therefore we consider two formalisms: first we provide a Logic Program with a similar structure as an $\mathcal{HPX}$-Logic Program and second we define an operational semantics with a similar structure as the $\mathcal{HPX}$ semantics. Then we present a soundness proof for the two formalisms.

Logic Program

Let LP_1 be the following Logic (sub-)Program:

$$q(X) \leftarrow u(X). \tag{A.3a}$$
$$q(X) \leftarrow q(Y), X \neq Y, v(X, Y). \tag{A.3b}$$
$$q(X) \leftarrow q(Y), X \neq Y, w(X, Y). \tag{A.3c}$$

where variables X and Y range over the set $\mathcal{X}$. Let S be a Stable Model of a LP that contains LP_1 and which does not have any other rules with a predicate q in the head, except those defined in LP_1. Then it follows by the Stable Model semantics that (A.4) is true.

$$\forall x, y \in \mathcal{X} : \big(q(x) \in S\big) \Leftarrow \big(u(x) \in S\big) \tag{A.4a}$$
$$\forall x, y \in \mathcal{X} : \big(q(x) \in S\big) \Leftarrow \big(\{q(y), v(x, y)\} \subseteq S \wedge x \neq y\big) \tag{A.4b}$$
$$\forall x, y \in \mathcal{X} : \big(q(x) \in S\big) \Leftarrow \big(\{q(y), w(x, y)\} \subseteq S \wedge x \neq y\big) \tag{A.4c}$$

The other direction of the implication (A.5) must also hold because the Logic Program does not contain any other rules with a $q/1$ predicate in the head.

$$\begin{aligned} \forall x, y \in \mathcal{X} : \Big[& q(x) \in S \Rightarrow \\ & \Big(\big(u(x) \in S\big) \\ & \vee \big(\{q(y), v(x, y)\} \subseteq S \wedge x \neq y\big) \\ & \vee \big(\{q(y), w(x, y)\} \subseteq S \wedge x \neq y\big)\Big)\Big] \end{aligned} \tag{A.5}$$

Set-theoretic Semantics

The set-theoretic semantics involves functions which set the properties of $\mathfrak{h}$. Assume that the following implications are defined by the set-theoretic semantics:

$$\forall x, y \in \mathcal{X} : p_{op}(x, \mathfrak{h}) \Leftarrow \big(u_{op}(x, \mathfrak{h})\big) \tag{A.6a}$$

$$\forall x, y \in \mathcal{X} : p_{op}(x, \mathfrak{h}) \Leftarrow \big(p_{op}(y, \mathfrak{h}) \wedge v_{op}(x, y, \mathfrak{h}) \wedge x \neq y\big) \tag{A.6b}$$

$$\forall x, y \in \mathcal{X} : p_{op}(x, \mathfrak{h}) \Leftarrow \big(p_{op}(y, \mathfrak{h}) \wedge w_{op}(x, y, \mathfrak{h}) \wedge x \neq y\big) \tag{A.6c}$$

A-priori relation between Logic Program and set-theoretical semantics

Assume that we have proven equivalence relations between the ASP implementation and the operational semantics as depicted in Table A.1.

ASP implementation		Operational semantics
$u(x) \in S$	$\Leftrightarrow$	$u_{op}(x, \mathfrak{h})$
$v(x, y) \in S$	$\Leftrightarrow$	$v_{op}(x, y, \mathfrak{h})$
$w(x, y) \in S$	$\Leftrightarrow$	$w_{op}(x, y, \mathfrak{h})$

Table A.1.: Relation between example Logic Program and example operational semantics

Lemma and Inductive Proof

A notion of soundness between the ASP implementation and the operational semantics is defined with respect to $q(x) \in S$ and $p_{op}(x, \mathfrak{h})$. The general form of a soundness Lemma is as follows:[3]

Lemma A.1 (Soundness lemma in general form)

$$\forall x \in \mathcal{X} : \big(q(x) \in S \Rightarrow p_{op}(x, \mathfrak{h})\big) \tag{A.7}$$

Proof:
The hypothesis is that (A.7) holds for arbitrary $x \in \mathcal{X}$. The relations depicted in Table A.1 allow us to perform a structural induction as follows.

Base step For the base step we consider those x for which $p_{op}(x)$ is produced by LP rule $A.3a$. To this end we substitute $q(x)$ with the body of A.4a and prove (A.8).

[3] The Lemma is a simplified version of implication (A.2), where we omit the $n + 1$ and b' parameters for simplicity.

$$\forall x \in \mathcal{X} : \big(u(x) \in S \Rightarrow p_{op}(\mathfrak{h}, x)\big) \tag{A.8}$$

It follows from Table A.1 that $u(x) \in S \Rightarrow u_{op}(x, \mathfrak{h})$. Therefore to show that (A.8) holds it is sufficient to show that (A.9) holds.

$$\forall x \in \mathcal{X} : \big(u_{op}(x, \mathfrak{h}) \Rightarrow p_{op}(\mathfrak{h}, x)\big) \tag{A.9}$$

It directly follows from (A.6a) that implication (A.9) holds.

Induction step 1 For the first induction step we consider those x for which $p_{op}(x)$ is produced by LP rule $A.3b$. To this end we substitute $q(x)$ with the body of A.4b and prove (A.10).

$$\forall x \in \mathcal{X} : \Big(\big(\{q(y), v(x,y)\} \subseteq S \wedge x \neq y\big) \Rightarrow p_{op}(\mathfrak{h}, x)\Big) \tag{A.10}$$

It follows from Table A.1 that $v(x,y) \in S \Rightarrow v_{op}(x, y, \mathfrak{h})$. Therefore to show that (A.10) holds it is sufficient to show that (A.11) holds.

$$\forall x \in \mathcal{X} : \Big(\big(q(y) \in\subseteq S \wedge x \neq y \wedge v_{op}(x, y, \mathfrak{h})\big) \Rightarrow p_{op}(\mathfrak{h}, x)\Big) \tag{A.11}$$

Note that we presume that $x \neq y$. For this reason we can use the induction hypothesis and assume that $q(y) \Rightarrow p_{op}(y, \mathfrak{h})$. Hence, to prove that (A.11) holds it is sufficient to show that (A.12) holds.

$$\forall x \in \mathcal{X} : \Big(\big(p_{op}(\mathfrak{h}, y) \wedge x \neq y \wedge v_{op}(x, y, \mathfrak{h})\big) \Rightarrow p_{op}(\mathfrak{h}, x)\Big) \tag{A.12}$$

According to (A.6b) implication (A.12) is clearly true.

Induction step 2 For the second induction step we consider those x for which $p_{op}(x)$ is produced by LP rule $A.3c$. That is, we show that (A.13) holds

$$\forall x \in \mathcal{X} : \Big(\big(\{q(y), w(X,Y)\} \subseteq S \wedge x \neq y\big) \Rightarrow p_{op}(\mathfrak{h}, x)\Big) \tag{A.13}$$

This case is analogous to induction step 1.

Completeness of induction The induction is complete because we have considered *all* rules in the Logic Program which can possibly produce an atom $q/1$. In other words, the induction is complete because (A.5) holds. ■

A.3. Soundness of Knowledge Atoms

Lemma A.2 is the main soundness lemma. It directly shows that the main soundness Theorem 4.1 from Section 4.6 holds.

Lemma A.2 (Soundness for knowledge atoms for single state transitions)

$$\begin{aligned} \forall n, b, b' : \quad & hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \exists \mathfrak{h} \in \Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) : \\ & \qquad \forall l, t : (knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l, t \rangle\,) \end{aligned} \tag{A.14}$$

For the proof of Lemma (A.2) we first distinguish whether or not $\exists l' : sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$. This makes it easy to argue under which circumstances $hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}})$ is true and eliminates the $\exists$-quantification over $\mathfrak{h}$. Presuming that $hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}})$ holds under certain circumstances and having eliminated the $\exists$-quantifications we perform induction over the structure of implications which produce pairs $\langle l, t \rangle$:

To this end, we perform several base steps to show that (A.14) holds for some $\langle l, t \rangle$. Then we perform induction steps where we show that that given (A.14) holds for some fixed $\langle l', t' \rangle$ it also holds for pairs $\langle l, t \rangle$ with $\langle l', t' \rangle \neq \langle l, t \rangle$.

We consider Lemma A.6 which identifies all ten rules in the $\mathcal{HPX}$-Logic Program that have a $knows/4$ predicate in their head and hence eventually produce a $knows(l, t, n+1, b')$ atom. As discussed in Lemma A.6, the ten LP rules correspond to implications (A.15) which are universally quantified over l, t, n, b'.

From these implications, (A.15a), (A.15f), (A.15j) and (A.15k) produce knowledge concerning pairs $\langle l, t \rangle$ independently from knowledge about other pairs $\langle l', t' \rangle$ for fixed n, b'. That is, to produce $knows(l, t, n+1, b')$ these implications do not directly depend on an atom $knows(l', t', n+1, b')$ in their body. For each of these rules we perform one base step.

The implications which cover initial state constraints (A.15b), (A.15c), forward inertia (A.15d), backward inertia (A.15e), causation (A.15g), positive postdiction (A.15h) and negative postdiction (A.15i) generate $knows(l, t, n+1, b')$ atoms dependently on $knows(l', t', n+1, b')$ atoms, where $\langle l', t' \rangle \neq \langle l, t \rangle$. We consider these rules for the induction steps where we may assume that soundness is given for $knows(l', t', n+1, b')$ atoms.

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(t=0 \wedge n=-1 \wedge b'=0 \wedge l \in \mathcal{VP}\Big) \tag{A.15a}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t=0 \wedge n=-1 \wedge b'=0 \wedge \\ &\qquad l \in \mathcal{C} \wedge \forall l^{+} \in \mathcal{C} \setminus l : knows(\overline{l^{+}},0,0,0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big)\end{aligned} \tag{A.15b}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t=0 \wedge n=-1 \wedge b'=0 \wedge \\ &\qquad \bar{l} \in \mathcal{C} \wedge \exists l^{+} \in \mathcal{C} \setminus \bar{l} \wedge knows(l^{+},0,0,0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big)\end{aligned} \tag{A.15c}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \big(knows(l,t-1,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ &\qquad kNotSet(\bar{l},t-1,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n+1\big)\end{aligned} \tag{A.15d}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \big(knows(l,t+1,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ &\qquad kNotSet(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t < n+1\big)\end{aligned} \tag{A.15e}$$

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow knows(l,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.15f}$$

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow kCause(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.15g}$$

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow kPosPost(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.15h}$$

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow kNegPost(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.15i}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \big(\exists b : sRes(l,n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t=n\big)\end{aligned} \tag{A.15j}$$

$$\begin{aligned}&knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ &\qquad \big(\exists b : (sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ &\qquad knows(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}) \wedge n \geq t\big)\end{aligned} \tag{A.15k}$$

The induction is complete because we consider *all* rules in the Logic Program which can eventually generate a $knows/4$-atom. That is, all possible $knows(l,t,n+1,b')$ atoms are reached for arbitrary n, b.

Proof:
(A.14)

Transition function (3.7):
$\Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \bigcup_{k \in sense(\mathbf{A},\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle)$
where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,t \rangle \,|a \in \boldsymbol{A}_{n,b} \wedge t = now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\}$

$$\begin{aligned} \forall n,b,b' : \quad & hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \exists \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A},\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \\ & \qquad \forall l,t : (knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l,t \rangle \,) \end{aligned} \tag{A.16}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,t \rangle \,|a \in \boldsymbol{A}_{n,b} \wedge t = now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\}$

According to Lemma A.13:
$now\big(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})\big) = n$

$$\begin{aligned} \forall n,b,b' : \quad & hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \exists \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A},\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \\ & \qquad \forall l,t : (knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l,t \rangle \,) \end{aligned} \tag{A.17}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle \,|a \in \boldsymbol{A}_{n,b}\}$

To prove that (A.17) holds, we perform induction for pairs $\langle l,t \rangle$. To this end, we first determine cases under which $hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}})$ becomes true and we eliminate the $\exists$ quantification over h-states. We distinguish two cases.

Case 1: $\exists l' : (sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}})$ (With Sensing Result)

We prove that (A.17) holds if sensing results are obtained. That is, we consider cases where (A.18) holds.

$$\exists l' : (sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \tag{A.18}$$

The following formulae are universally quantified over those n,b,b' for which (A.18) holds. The case distinction allows us to simplify (A.17) and we make the following substitutions for this case.

Recall (A.17):

$$hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow$$
$$\exists \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k) :$$
$$\forall l, t : (knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l, t \rangle\,)$$

Case distinction (A.18) and definition of $hasChild$ (4.3):

$\exists l' : (sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}})$

$$\exists \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k) : \quad \forall l, t : (knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l, t \rangle\,) \tag{A.19}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$

We eliminate the $\exists$-quantification by generalizing (A.19) as follows:
The case distinction (A.18) states that there exists at least one literal l' for which $sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$. In the following we consider an arbitrary literal l^s such that:

$$sRes(l^s, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.20}$$

That is, we presume that the following formulae are implicitly universally quantified over l^s for which (A.20) holds.
Further, according to Lemma A.10:

$$\Big(sRes(l^s, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \Rightarrow \Big(sense(\mathbf{A}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \big\{\{\langle l^s, n\rangle\}, \{\langle \overline{l^s}, n\rangle\}\big\}\Big)$$

Hence, in order to show that (A.19) holds, it is sufficient to show that (A.21) holds.

$$\forall l, t : \Big(\big(knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}\big) \Rightarrow \big(eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n \rangle) \models \langle l, t \rangle\,\big)\Big) \tag{A.21}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$

We prove (A.21) by induction over the structure of implications (A.15) which generate $knows/4$ atoms for pairs $\langle l, t \rangle$.

Base Steps for Case 1

1. *Initial Knowledge:* $\{\langle l,t\rangle \mid knows(l,t,n+1,b')$ is produced by (T2)$\}$
   Implication (A.15a) generates knowledge for step 0 only, i.e. according to (A.15a) it must hold that if an atom $knows(l,t,n+1,b')$ is generated then $n+1=0$. However, since by Definition 4.1 we consider $n \geq 0$, (T2) can not produce an atom $knows(l,t,n+1,b)$; this case does not apply.

2. *Inertia of knowledge:* $\{\langle l,t\rangle \mid knows(l,t,n+1,b)$ is produced by (F3f)$\}$
   Recall the LP rule (F3f):

$$knows(L,T,N,B) \leftarrow knows(L,T,N-1,B), N \leq \texttt{maxS}.$$

In the following we show that (A.21) holds for $knows(l,t,n+1,b')$ produced by (F3f).

Recall (A.21):

$$\forall l,t : (knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s,n\rangle\rangle) \models \langle l,t\rangle)$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle \mid a \in \boldsymbol{A}_{n,b}\}$

To prove (A.21) for those $\langle l,t\rangle$ for which an atom $knows(l,t,n+1,b')$ is produced by Logic Programming rule (F3f) we consider the following implication (A.15f):

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow knows(l,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$$

We substitute $knows(l,t,n+1,b')$ in (A.21) with the body of (A.15f) and obtain (A.22).

$$knows(l,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})\rangle) \models \langle l,t\rangle \tag{A.22}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle \mid a \in \boldsymbol{A}_{n,b}\}$

By Lemma A.4:

$$\forall l,t,n,b,b' : \big(knows(l,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge (\exists l' : sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}})\big) \Rightarrow b = b'$$

According to case distinction (A.18) we consider only cases where $b = b'$.

Consequently we consider only those cases where $b = b'$. In theses cases the rest of the proof is analogous to the Case 1 where $\neg\exists l', b' : sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$. (In that case it also holds that $b = b'$.)

3. *Sensing* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F5k)$\}$
   Recall the LP rule (F5k):

$$knows(L,N-1,N,B') \leftarrow sRes(L,N-1,B,B'),s(N)$$

In the following we show that (A.21) holds for those $knows(l,t,n+1,b')$ produced by (F5k).

Recall (A.21):

$$\forall l,t : (knows(l,t,n+1,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s,n\rangle\rangle) \models \langle l,t\rangle)$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

We prove (A.21) for those $\langle l,t\rangle$ for which an atom $knows(l,t,n+1,b')$ is produced by Logic Programming rule (F5k), respectively by the following implication (A.15j):

$$knows(l,t,n+1,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Leftarrow (sRes(l,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge t=n)$$

We substitute $knows(l,t,n+1,b')$ in (A.21) with the body of (A.15j) and obtain (A.24).

$$\forall l,t : sRes(l,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s,t\rangle\rangle) \models \langle l,t\rangle \tag{A.23}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Recall that by (A.20), l^s is an arbitrary literal such that $sRes(l^s,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$.
By Lemma A.11:

$$(sRes(l,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge sRes(l^s,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}) \Rightarrow l = l^s$$

That is, in the following we consider $l = l^s$.

$$\forall t : sRes(l^s,n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s,t\rangle\rangle) \models \langle l^s,t\rangle \tag{A.24}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

By Lemma B.7:

$$eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s,t\rangle\rangle) \models \langle l^s,t\rangle$$

The base step is proven for knowledge about $\langle l,n\rangle$ generated by rule (F5k) (sensing) where $\exists l' : sRes(l',n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$.

4. *Inheritance* $\{\langle l,t\rangle\,|knows(l,t,n+1,b')$ is produced by (F5m)$\}$
   Recall the LP rule (F5m):

$$knows(L,T,N,B') \leftarrow sRes(_,N-1,B,B'), neq(B,B'), knows(L,T,N-1,B), N \geq T$$

In the following we show that (A.21) holds for those $knows(l,t,n+1,b')$ produced by (F5k).

Recall (A.21):

$$\forall l,t: \Big(\big(knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}}\big) \Rightarrow \big(eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s,n\rangle) \models \langle l,t\rangle\,\big)\Big)$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle\,|a \in \boldsymbol{A}_{n,b}\}$

We prove (A.21) for those $\langle l,t\rangle$ for which an atom $knows(l,t,n+1,b')$ is produced by Logic Programming rule (F5m), respectively by the following implication (A.15k):

$$knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \big(\exists l': \{sRes(l',n,b,b'), knows(l,t,n,b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n+1 \geq t\big)$$

We substitute $knows(l,t,n+1,b')$ in (A.21) with the body of (A.15j) and obtain (A.25).

$$\forall l,t: \Big(\big(\exists l': \{sRes(l',n,b,b'), knows(l,t,n,b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n+1 \geq t\big) \Rightarrow \big(eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s,n\rangle) \models \langle l,t\rangle\,\big)\Big) \tag{A.25}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle\,|a \in \boldsymbol{A}_{n,b}\}$

With (4.5): $\forall \langle l,t\rangle : knows(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \models \langle l,t\rangle$

We consider $t \leq n$ anyways and can ignore the term $n+1 \geq t$.

$$\forall l,t: \Big(\big(\exists l': sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \models \langle l,t\rangle\,\big) \Rightarrow \big(eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s,n\rangle) \models \langle l,t\rangle\,\big)\Big) \tag{A.26}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle\,|a \in \boldsymbol{A}_{n,b}\}$

By Lemma B.7: $\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow eval(\boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) \models \langle l,t\rangle$

Hence, it is sufficient to show that (A.27) holds.

$$\forall l,t : \Big(\big(\exists l' : sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \models \langle l,t\rangle\big) \Rightarrow \big(\langle l,t\rangle \in \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})\big)\Big) \tag{A.27}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

With (4.5):
$\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \models \langle l,t\rangle \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})$

The base step is proven for knowledge generated by rule (F5m) (inheritance).

Induction Steps for Case 1

1. *Initial State Constraints:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b')$ is produced by (T3)$\}$
Implications (A.15b) and (A.15c) generates knowledge for step 0 only, i.e. if an atom $knows(l,t,n+1,b')$ is generated then $n+1=0$. However, since by Definition 4.1 we consider $n \geq 0$, (T2) can not produce an atom $knows(l,t,n+1,b)$; this case does not apply.

2. *Forward inertia:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F3d)$\}$
Recall the Logic Programming rule (F3d):

$$knows(L,T,N,B) \leftarrow knows(L,T-1,N,B), \\ kNotSet(\overline{L},T-1,N,B), complement(L,\overline{L}), T \leq N.$$

In the following we show that (A.19) holds for those $knows(l,t,n+1,b')$ which are produced by (F3d).

Recall (A.21):

$$\forall l,t : (knows(l,t,n+1,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s,n\rangle\rangle) \models \langle l,t\rangle)$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

We prove (A.19) for those $\langle l, t\rangle$ for which an atom $knows(l, t, n+1, b')$ is produced by Logic Programming rule (F3d), respectively by the following implication (A.15d):

$$knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ \{knows(l, t-1, n+1, b'), kNotSet(\bar{l}, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n$$

We substitute $knows(l, t, n+1, b')$ in (A.21) with the body of (A.15d) and obtain (A.28).

$$\{knows(l, t-1, n+1, b'), kNotSet(\bar{l}, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n \Rightarrow \\ eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle) \models \langle l, t\rangle \tag{A.28}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

We can eliminate $t \leq n$ since this is considered anyways. Further, recall that by the recursive definition of $eval$ (3.17):

$$eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle) = \\ evalOnce(eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle))$$

$$\{knows(l, t-1, n+1, b'), kNotSet(\bar{l}, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \\ evalOnce(\mathfrak{h}') \models \langle l, t\rangle \tag{A.29}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Considering $evalOnce(\mathfrak{h}') = pd^{neg}(pd^{pos}(cause(back(fwd(\mathfrak{h}')))))$ (B.2) and its constituent functions (3.11), (3.12), (3.13), (3.14), (3.15), in particular add_{fwd} (3.11), it follows that:

$$\langle l, t\rangle \in add_{fwd}(\mathfrak{h}') \Rightarrow evalOnce(\mathfrak{h}') \models \langle l, t\rangle$$

Hence, to show that (A.29) holds, it is sufficient to show that (A.30) holds:

$$\{knows(l, t-1, n+1, b'), kNotSet(\bar{l}, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \\ \langle l, t\rangle \in add_{fwd}(\mathfrak{h}') \tag{A.30}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Consider Lemma A.3:

$$\forall l, l', t, n, b', l^s : \Big(\{kNotSet(l, t-1, n, b'), sRes(l^s, t-1, n, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ \big(knows(l', t-1, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l', t-1 \rangle \in \boldsymbol{\kappa}' \big) \Big) \\ \Rightarrow inertial(\bar{l}, t-1, \mathfrak{h}')$$

Due to the induction hypothesis we may assume that $\big(knows(l', t-1, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l', t-1 \rangle \in \boldsymbol{\kappa}'\big)$ is true, and hence we can substitute $kNotSet(l, t-1, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}$ with $inertial(l, t-1, \mathfrak{h}')$.

$$knows(l, t-1, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge inertial(l, t-1, \mathfrak{h}') \Rightarrow \\ \langle l, t \rangle \in add_{fwd}(\mathfrak{h}') \tag{A.31}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n \rangle \rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$

Again, the induction hypothesis allows us to assume that:

$$knows(l, t-1, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \\ \Rightarrow \langle l, t-1 \rangle \in \boldsymbol{\kappa}(\mathfrak{h}')$$

$$\langle l, t-1 \rangle \in \boldsymbol{\kappa}(\mathfrak{h}') \wedge inertial(l, t-1, \mathfrak{h}') \Rightarrow \\ \langle l, t \rangle \in add_{fwd}(\mathfrak{h}') \tag{A.32}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l', n \rangle \rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$

Consider the definition of add_{fwd} (3.11):

$$add_{fwd}(\mathfrak{h}') = \{\langle l, t \rangle \,|\, \langle l, t-1 \rangle \in \boldsymbol{\kappa}(\mathfrak{h}') \wedge inertial(l, t-1, \mathfrak{h}') \wedge t \leq now(\mathfrak{h}')\}$$

By the definition of now (3.5), the definition of $\boldsymbol{A}_{n,b}$ (see Definition 4.1) and the fact that $\neg\exists \langle a, t \rangle \in \boldsymbol{\alpha}' : t > n$ (see (4.5b) it holds that

$$now(\mathfrak{h}') = n + 1$$

Since We have shown for Case 2 (A.18) that if knowledge is produced by forward inertia rule (F3d) the soundness Lemma (A.14) holds.

3. *Backward inertia:* $\{\langle l, t \rangle \,| knows(l, t, n+1, b)$ is produced by (F3e)$\}$
   This case is analogous to the case of forward inertia.

4. *Causation:* $\{\langle l, t\rangle \,| knows(l, t, n+1, b)$ is produced by (F4a)$\}$
   Recall (F4a):

$$knows(L, T, N, B) \leftarrow kCause(L, T, N, B)$$

We show that (A.19) holds for those $knows(l, t, n+1, b')$ which are generated by (F4a).

Recall (A.19):

$$\forall l, t : (knows(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle) \models \langle l, t\rangle)$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a, n\rangle \,| a \in \boldsymbol{A}_{n,b}\}$

We prove (A.19) for those $\langle l, t\rangle$ for which an atom $knows(l, t, n+1, b')$ is produced by Logic Programming rule (F4a), respectively by the following implication (A.15g):

$$knows(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Leftarrow kCause(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \tag{A.33}$$

Since $kCause/4$ atoms are only produced by LP rules generated by translation rule (T6a) we have according to (T6a):

$$\begin{aligned} &kCause(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Leftrightarrow \\ &\quad \exists ep : \big(e(ep) = l \wedge c(ep) = \{l^c_1, \ldots, l^c_k\} \wedge apply(ep, t-1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge n \geq t \wedge \\ &\qquad \{knows(l^c_1, t-1, n+1, b'), \ldots, knows(l^c_k, t-1, n+1, b')\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}}\big) \end{aligned} \tag{A.34}$$

To show that (A.19) holds for those $\langle l, t\rangle$ produced by LP rule (F4a) we prove (A.35).

$$kCause(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle) \models \langle l, t\rangle \tag{A.35}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a, n\rangle \,| a \in \boldsymbol{A}_{n,b}\}$

Due to the inductive definition of $eval$ (3.17):

$$\begin{aligned} &eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle) = \\ &\qquad evalOnce(eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle)) \end{aligned}$$

$$kCause(l, t, n+1, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow evalOnce(\mathfrak{h}') \models \langle l, t\rangle \tag{A.36}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a, n\rangle \,| a \in \boldsymbol{A}_{n,b}\}$

Considering $evalOnce(\mathfrak{h}') = pd^{neg}(pd^{pos}(cause(back(fwd(\mathfrak{h}')))))$ (B.2) and its constituent functions (3.11), (3.12), (3.13), (3.14), (3.15) it follows that

$$\langle l, t\rangle \in add_{cause}(\mathfrak{h}') \Rightarrow evalOnce(\mathfrak{h}') \models \langle l, t\rangle$$

Hence, to show that (A.36) holds, it is sufficient to show that (A.37) holds:

$$kCause(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l, t\rangle \in add_{cause}(\mathfrak{h}') \tag{A.37}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Consider the equivalence for $kCause/4$ atoms (A.34):

$$\begin{aligned} &kCause(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow \\ &\quad \exists ep : \big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge apply(ep, t-1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t \wedge \\ &\qquad\quad \{knows(l_1^c, t-1, n+1, b'), \ldots, knows(l_k^c, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \end{aligned}$$

$$\begin{aligned} &\exists ep : \big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge apply(ep, t-1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t \wedge \\ &\qquad \{knows(l_1^c, t-1, n+1, b'), \ldots, knows(l_k^c, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \\ &\Rightarrow \langle l, t\rangle \in add_{cause}(\mathfrak{h}') \end{aligned} \tag{A.38}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

By the induction hypothesis:

$$\{knows(l_1^c, t-1, n+1, b'), \ldots, knows(l_k^c, t-1, n+1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}$$
$$\Rightarrow \{\langle l_1^c, t-1\rangle, \ldots, \langle l_n^c, t-1\rangle\} \subseteq \boldsymbol{\kappa}(\mathfrak{h}')$$

$$\begin{aligned}&\exists ep : \big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge apply(ep, t-1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t \wedge \\ &\qquad \{\langle l_1^c, t-1\rangle, \ldots, \langle l_n^c, t-1\rangle\} \subseteq \boldsymbol{\kappa}(\mathfrak{h}')\big) \\ &\Rightarrow \langle l, t\rangle \in add_{cause}(\mathfrak{h}')\end{aligned} \tag{A.39}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

By Lemma A.8:

$$(apply(ep, t-1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge sRes(l^s, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t-1 \leq n) \quad \Rightarrow apply(ep, t-1, b)$$

By Lemma A.7:

$$apply(ep, t-1, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}')$$

$$\begin{aligned}&\exists ep : \big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge \langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') \wedge n \geq t \wedge \\ &\qquad \{\langle l_1^c, t-1\rangle, \ldots, \langle l_n^c, t-1\rangle\} \subseteq \boldsymbol{\kappa}(\mathfrak{h}')\big) \\ &\Rightarrow \langle l, t\rangle \in add_{cause}(\mathfrak{h}')\end{aligned} \tag{A.40}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Consider the definition of add_{cause} (3.13):

$$\begin{aligned}add_{cause}(\mathfrak{h}') = \{\langle l, t\rangle \,|\exists \langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') :\ & \\ &\big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge \\ &\{\langle l_1^c, t-1\rangle, \ldots, \langle l_n^c, t-1\rangle\} \subseteq \boldsymbol{\kappa}(\mathfrak{h}')\big)\}\end{aligned}$$

We have shown for Case 2 (A.18) that if knowledge is produced by causation rule (F4a) the soundness Lemma (A.14) holds.

5. *Positive postdiction:* $\{\langle l, t\rangle \,|knows(l, t, n+1, b)$ is produced by (F4b)$\}$
   This case is analogous to the case of causation.

6. *Negative postdiction:* $\{\langle l, t\rangle \,|knows(l, t, n+1, b)$ is produced by (F4c)$\}$

This case is analogous to the case of causation.

Case 2: $\neg\exists l' : (sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}})$ (No Sensing Result)

We prove that (A.17) holds if no sensing results are obtained. Therefore we consider cases where (A.41) holds.

$$\neg\exists l' : (sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \tag{A.41}$$

That is, the following formulae are universally quantified over those n, b, b' for which (A.41) holds. With the case distinction (A.17) can be simplified further as follows:

Recall (A.17):

$$\begin{aligned}\forall n, b, b' : \quad & hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \exists \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \\ & \qquad \forall l, t : (knows(l, t, n+1, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \mathfrak{h} \models \langle l, t \rangle)\end{aligned}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

Case distinction (A.41) and Lemma A.10:

$$sense(\mathbf{A}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\emptyset\}$$

Case distinction (A.41) and definition of $hasChild$ (4.3):

$$hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Leftrightarrow b = b'$$

$$\forall l, t : (knows(l, t, n+1, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \rangle) \models \langle l, t \rangle) \tag{A.42}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

We prove (A.42) by induction over pairs $\langle l, t \rangle$.

Base Steps for Case 2

1. *Initial Knowledge:* $\{\langle l, t \rangle \, | knows(l, t, n+1, b)$ is produced by (T2)$\}$
   This is analogous to Case 1.

2. *Inertia of knowledge:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F3f)$\}$

   Recall (A.42):

   $$\forall l,t : (knows(l,t,n+1,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}})\rangle) \models \langle l,t\rangle)$$

   with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

   We prove (A.19) for those $\langle l,t\rangle$ for which an atom $knows(l,t,n+1,b')$ is produced by Logic Programming rule (F3f), respectively by the following implication (A.15f):

   $$knows(l,t,n+1,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Leftarrow knows(l,t,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}$$

   We substitute $knows(l,t,n+1,b')$ in (A.42) with the body of (A.15f) and obtain (A.43).

   $$knows(l,t,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}})\rangle) \models \langle l,t\rangle \tag{A.43}$$

   with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

   By Lemma B.7:

   $$\langle l,t\rangle \in \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}})\rangle) \models \langle l,t\rangle$$

   $$knows(l,t,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \tag{A.44}$$

   with $t \leq n$.

   By (4.5):

   $$\boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) = \{\langle l,t\rangle \,|knows(l,t,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}\}$$

   We have shown for Case 1 (A.41) that the soundness Lemma (A.14) holds for those $\langle l,t\rangle$ for which $knows(l,t,n+1,b)$ is produced by inertia of knowledge (F3f).

3. *Sensing* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F5k)$\}$
   If an atom $knows(l,t,n+1,b)$ is generated by sensing, then according to Lemma A.6, (A.60j) it must hold that $\exists l' : sRes(l,n,b,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge t = n$. Since this contradicts the the case distinction (A.41) this case does not apply.

4. *Inheritance* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F5m)$\}$
   If an atom $knows(l,t,n+1,b)$ is generated by inheritance, then according to Lemma A.6, (A.60k) it must hold that $\exists l' : sRes(l',n,b,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}$. Since this

contradicts the the case distinction (A.41) this case does not apply.

Induction Steps for Case 2

1. *Initial state constraints:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (T3)$\}$
   This is analogous to Case 1.

2. *Forward inertia:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F3d)$\}$
   This is analogous to Case 1.

3. *Backward inertia:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F3e)$\}$
   This is analogous to the case of backward inertia.

4. *Causation:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F4a)$\}$
   This is analogous to Case 1.

5. *Positive postdiction:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F4b)$\}$
   This is analogous to the case of causation.

6. *Negative postdiction:* $\{\langle l,t\rangle \,|knows(l,t,n+1,b)$ is produced by (F4c)$\}$
   This is analogous to the case of causation.

■

Auxiliary Lemmata Related to Knowledge

Inertia

The following Lemma is required in the induction step for proving soundness of forward inertia (A.28) in Section A.3.

Lemma A.3 (Soundness for inertia in induction step)

$$\begin{aligned} \forall l, l', t, n, b', l^s : \Big(& \{kNotSet(l,t,n,b'), sRes(l^s,t,n,b')\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}} \wedge \\ & \big(knows(l',t,n,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow \langle l', t\rangle \in \boldsymbol{\kappa}'\big)\Big) \\ & \Rightarrow inertial(\bar{l}, t, \mathfrak{h}') \end{aligned} \tag{A.45}$$

with $t \leq n$ *and* $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \langle l^s, n\rangle\rangle)$ *where* $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}}) \cup \{\langle a,n\rangle \, | a \in \boldsymbol{A}_{n,b}\}$.

Proof:
Consider the rules in an $\mathcal{HPX}$-LP which trigger $kNotSet/4$ atoms:

$$kNotSet(L,T,N,B) \leftarrow not\ kMaySet(L,T,B), uBr(N,B), s(T), literal(L). \tag{F3a}$$

$$kMaySet(L,T,B) \leftarrow apply(EP,T,B), hasEff(EP,L) \tag{F3b}$$

$$\begin{aligned} kNotSet(L,T,N,B) \leftarrow & apply(EP,T,B), hasCond(EP,L'), hasEff(EP,L), \\ & knows(\overline{L'},T,N,B), complement(L',\overline{L'}), N >= T. \end{aligned} \tag{F3c}$$

Since (F3a) and (F3c) are the only rules in the LP with $kNotSet/4$ in their head, it holds that:

$$\begin{aligned} & kNotSet(l,t,n,b) \\ & \Leftrightarrow \\ & \Big(\big(kMaySet(l,t,b) \notin S^{\boldsymbol{P}}_{\mathcal{D}} \wedge \{uBr(n,b), s(t), literal(l)\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}}\big) \\ & \vee \big(\exists ep, l^c : \big(apply(ep,t,b), hasCond(ep,l^c), hasEff(ep,l), knows(\overline{l^c},t,n,b) \subseteq S^{\boldsymbol{P}}_{\mathcal{D}} \\ & \qquad \wedge n >= t\big)\big)\Big) \end{aligned} \tag{A.46}$$

We make a case distinction according to (A.46) and consider both possibilities which can trigger an atom $kNotSet(l,t,n,b)$.

1. $kNotSet/4$ generated by (F3a)
   In this case (A.47) holds and we prove (A.48)

$$\big(kMaySet(l,t,b) \notin S^{\boldsymbol{P}}_{\mathcal{D}} \wedge \{uBr(n,b), s(t), literal(l)\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}}\big) \tag{A.47}$$

$$
\begin{aligned}
&\forall l, l', t, n, b', l^s : \\
&\Big(sRes(l^s, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\
&\big(kMaySet(l, t, b) \notin S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \{uBr(n, b), s(t), literal(l)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \wedge \\
&\big(knows(l', t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l', t\rangle \in \boldsymbol{\kappa}'\big)\Big) \\
&\Rightarrow inertial(\bar{l}, t, \mathfrak{h}')
\end{aligned}
\tag{A.48}
$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$.

Consider rule (F3b). This is the only rule in the LP with $kMaySet/3$ in the head and therefore it holds that:

$$
kMaySet(l, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow \exists ep : (\{apply(ep, t, b), hasEff(ep, l)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}) \tag{A.49}
$$

With rewrite (A.49) as follows:

$$
kMaySet(l, t, b) \notin S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow \forall ep : (apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow hasEff(ep, l) \notin S_{\mathcal{D}}^{\boldsymbol{P}})
$$

$$
\begin{aligned}
&\forall l, l', t, n, b', l^s : \\
&\Big(sRes(l^s, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\
&\big(\forall ep : (apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow hasEff(ep, l) \notin S_{\mathcal{D}}^{\boldsymbol{P}})\big) \\
&\big(\{uBr(n, b), s(t), literal(l)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \\
&\big(knows(l', t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l', t\rangle \in \boldsymbol{\kappa}'\big)\Big) \\
&\Rightarrow inertial(\bar{l}, t, \mathfrak{h}')
\end{aligned}
\tag{A.50}
$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n\rangle\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$.

Consider the definition of $inertial$ (3.10):

$$
\begin{aligned}
&inertial(\bar{l}, t, \mathfrak{h}') \Leftrightarrow \\
&\quad \forall \langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') : \\
&\quad (e(ep) = l) \Rightarrow \big(\exists l^c \in c(ep) : \langle \overline{l^c}, t\rangle \in \boldsymbol{\kappa}'\big)
\end{aligned}
$$

We have shown that (A.48) holds.

2. $kNotSet/4$ generated by (F3c)
   In this case (A.51) holds and we prove (A.52)

$$\exists ep, l' : (n >= t \wedge \big(apply(ep,t,b), hasCond(ep,l'), hasEff(ep,l), knows(\overline{l'},t,n,b) \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \tag{A.51}$$

$$\begin{aligned}
&\forall l, l', t, n, b', l^s : \\
&\Big(sRes(l^s,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\
&\big(\exists ep, l' : (\{apply(ep,t,b'), hasEff(ep,l)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \wedge n >= t \\
&\qquad \{hasCond(ep,l'), knows(\overline{l'},t,n,b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}})\big) \wedge \\
&\big(knows(l',t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l',t \rangle \in \boldsymbol{\kappa}'\big)\Big) \\
&\Rightarrow inertial(\bar{l},t,\mathfrak{h}')
\end{aligned} \tag{A.52}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n \rangle \rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle | a \in \boldsymbol{A}_{n,b}\}$.

Consider LP rule (2) which restricts that two effect propositions with the same effect literal may not be applied concurrently:

$$\begin{aligned}
\leftarrow\ & apply(EP_1,T,B), hasEff(EP_1,L), apply(EP_2,T,B), hasEff(EP_2,L), \\
& EP_1 \neq EP_2, br(B), literal(L).
\end{aligned}$$

By the definition of integrity constraints in ASP (which we described in Section 2.2.4) it follows that the following holds:

$$\begin{aligned}
\forall l : \big(\exists ep, l^c : \big(&\{apply(ep,t,b'), hasEff(ep,l)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\
&\{hasCond(ep,l^c), knows(\overline{l^c},t,n,b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}})\big) \\
\Rightarrow \big(\forall ep : \big(&(apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge hasEff(ep,l) \in S_{\mathcal{D}}^{\boldsymbol{P}}) \\
&\Rightarrow \big(\exists l^c : \{hasCond(ep,l^c), knows(\overline{l^c},t,n,b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}})\big)\big)
\end{aligned}$$

$$\begin{aligned}
\forall l, t, n, b', l^s : \Big(&sRes(l^s,t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\
&\big(\forall ep : \big((apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge hasEff(ep,l) \in S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\
&\qquad \big(\exists l^c : \{hasCond(ep,l^c), knows(\overline{l^c},t,n,b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}})\big)\big) \wedge \\
&\big(knows(l',t,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle l',t \rangle \in \boldsymbol{\kappa}'\big)\Big) \\
&\Rightarrow inertial(\bar{l},t,\mathfrak{h}')
\end{aligned} \tag{A.53}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n \rangle \rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle | a \in \boldsymbol{A}_{n,b}\}$.

Consider Lemma A.7:

$$\forall ep, t : apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \langle ep, t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h}')$$

An Lemma A.12:

$$\begin{aligned} hasCond(ep, l^c) \in S_{\mathcal{D}}^{\boldsymbol{P}} &\Rightarrow l^c \in c(ep) \\ hasEff(ep, l) \in S_{\mathcal{D}}^{\boldsymbol{P}} &\Rightarrow l = e(ep) \end{aligned}$$

$$\begin{aligned} hasCond(ep, l^c) \in S_{\mathcal{D}}^{\boldsymbol{P}} &\Rightarrow l^c \in c(ep) \\ hasEff(ep, l) \in S_{\mathcal{D}}^{\boldsymbol{P}} &\Rightarrow l = e(ep) \end{aligned}$$

It further holds that:

$$\left(knows(\overline{l^c}, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \left\langle \overline{l^c}, t \right\rangle \in \boldsymbol{\kappa}'\right)$$

To show that (A.53) holds, it is sufficient to show that (A.54) holds.

$$\begin{aligned} \forall l, t, n, b', l^s : \Big(& sRes(l^s, t, n, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ & \big(\forall ep : \big((\langle ep, t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') \wedge \Rightarrow l = e(ep)) \Rightarrow \\ & \qquad \big(\exists l^c : l^c \in c(ep) \wedge \left\langle \overline{l^c}, t \right\rangle \in \boldsymbol{\kappa}'\big)\big)\big)\Big) \wedge \\ & \Rightarrow inertial(\bar{l}, t, \mathfrak{h}') \end{aligned} \tag{A.54}$$

with $t \leq n$ and $\mathfrak{h}' = eval(\left\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \langle l^s, n \rangle \right\rangle)$ where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$.

Consider the definition of $inertial$ (3.10):

$$\begin{aligned} & inertial(\bar{l}, t, \mathfrak{h}') \Leftrightarrow \\ & \quad \forall \langle ep, t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') : \\ & \quad (e(ep) = l) \Rightarrow \left(\exists l^c \in c(ep) : \left\langle \overline{l^c}, t \right\rangle \in \boldsymbol{\kappa}'\right) \end{aligned}$$

We have shown that (A.52) holds.

■

No Knowledge in New Branches

The following Lemma is required in Section A.3, in the base step of the inductive soundness proof, where it is shown that soundness holds for knowledge generated by inheritance.

Lemma A.4 (Knowledge does not exist in new branches) :

$$\forall l,t,n,b,b' : \big(knows(l,t,n,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge (\exists l' : sRes(l',n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}})\big) \rightarrow b = b' \tag{A.55}$$

Proof Sketch:

Consider the following integrity constraint (F5i) in the domain independent theory of an $\mathcal{HPX}$-Logic Program which prevents that $sRes/4$ are produced in unused branches.

$$\leftarrow sRes(L,N,B,B'), uBr(N,B'), literal(L), neq(B,B') \tag{F5i}$$

It follows from the integrity constraint (F5i) that

$$(\exists l' : sRes(l',n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge b \neq b') \Rightarrow uBr(n,b') \notin S^{\boldsymbol{P}}_{\mathcal{D}} \tag{A.56}$$

Hence we can rewrite (A.55) as follows:

$$\forall l,t,n,b,b' : \big(knows(l,t,n,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow uBr(n,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}\big) \tag{A.57}$$

Lemma A.5 shows that (A.57) holds. This proves the Lemma. ■

No Knowledge in Unused Branches

The following Lemma is required to prove Lemma A.4.

Lemma A.5 (Knowledge does not exist in unused branches) :

$$\forall l,t,n,b : \big(knows(l,t,n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow uBr(n,b) \in S^{\boldsymbol{P}}_{\mathcal{D}}\big) \tag{A.58}$$

Proof:
This To show that (A.58) is true, we go through all 10 LP rules that generate a $knows/4$ atom. These are summarized as (bi-)implications in Lemma A.6, Equation (A.60). We prove (A.58) by complete induction over the structure of (A.60) for l,t,n. For the base steps, we show that for some of the implications in (A.60) it holds that (A.58) is true. For the induction steps we show that if (A.58) holds for certain triples $\langle l,t,n\rangle$ then it holds for other triples $\langle l',t',n'\rangle$ with $\langle l,t,n\rangle \neq \langle l',t',n'\rangle$. The induction is complete because we consider *all* implications in (A.60).

Base Steps

1.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(t = 0 \wedge n = 0 \wedge b = 0 \wedge l \in \mathcal{VP}\Big) \tag{A.59a}$$

This is true since by LP fact (F5a) it holds that $uBr(0,0) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

2.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t = 0 \wedge n = 0 \wedge b = 0 \wedge l \in \mathcal{C} \wedge \forall l^{+} \in \mathcal{C} \setminus l : knows(\overline{l^{+}}, 0, 0, 0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big)$$

This is true since by LP fact (F5a) it holds that $uBr(0,0) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

3.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t = 0 \wedge n = 0 \wedge b = 0 \wedge \bar{l} \in \mathcal{C} \wedge \exists l^{+} \in \mathcal{C} \setminus \bar{l} \wedge knows(l^{+}, 0, 0, 0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big) \tag{A.59b}$$

This is true since by LP fact (F5a) it holds that $uBr(0,0) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

4.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(sRes(l, n-1, b', b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \tag{A.59c}$$

This is follows directly from LP rule (F5j).

5.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(\{sRes(l', n-1, b', b), knows(l, t, n-1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t\Big) \tag{A.59d}$$

This is follows directly from LP rule (F5j).

Induction Steps

1.

$$uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \Big(\{knows(l, t-1, n, b), kNotSet(\bar{l}, t-1, n, b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n\Big) \tag{A.59e}$$

This is true since by induction hypothesis it holds that $knows(l, t-1, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

2.

$$\begin{aligned} & uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ & \Big(\{knows(l,t+1,n,b), kNotSet(\bar{l},t-1,n,b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n\Big) \end{aligned} \tag{A.59f}$$

This is true since by induction hypothesis it holds that $knows(l,t+1,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

3.

$$\begin{aligned} & uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ & Big(knows(l,t,n-1,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \end{aligned} \tag{A.59g}$$

This is true since by induction hypothesis it holds that $\{knows(l,t,n-1,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n-1,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$. By LP rule (F5c) it holds that $uBr(n-1,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$.

4.

$$\begin{aligned} & uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ & Big(kCause(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \end{aligned} \tag{A.59h}$$

By translation rule (T6a) and by considering that LP rules generated by (T6a) are the only LP rules with $kCause/4$ in the head it holds that

$$\begin{aligned} & kCause(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow \\ & \Big(\exists ep : \big(e(ep) = l \wedge c(ep) = \{l_1^c, \ldots, l_k^c\} \wedge \\ & \qquad apply(ep,t-1,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n > t \wedge \\ & \qquad \{knows(l_1^c,t-1,n,b), \ldots, knows(l_k^c,t-1,n,b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big) \end{aligned}$$

Since we can assume by induction hypothesis that for all $i \in \{1,\ldots,k\}$: $knows(l_i^c,t-1,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$ it must hold that (A.59h) is true.

5.

$$\begin{aligned} & uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ & \Big(kCause(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \end{aligned} \tag{A.59i}$$

This case is similar to (A.59h)

6.

$$\begin{aligned} & uBr(n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \\ & \Big(kCause(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \end{aligned} \tag{A.59j}$$

This case is similar to (A.59h)

■

Occurrence of $knows/4$ Atoms in a Stable Model

The following Lemma represents a bi-implication which states the necessary and required set-theoretic conditions under which an atom $knows(l,t,n,b)$ can be contained in the Stable Model of an $\mathcal{HPX}$-Logic Program.

Lemma A.6 (Knowledge generation in an $\mathcal{HPX}$-Logic Program) *Consider the notational conventions from Definition 4.1, i.e. we have a domain $\mathcal{D} = \langle \mathcal{VP}, \mathcal{ISC}, \mathcal{A}, \mathcal{G} \rangle$ and a set of atoms $\boldsymbol{P}$ denoting the occurrence of actions, such that $S_{\mathcal{D}}^{\boldsymbol{P}}$ is a Stable Model of $LP(\mathcal{D}) \cup \boldsymbol{P}$. Then the following equivalence holds:*

$$knows(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow$$

$$\Big[\Big(t = 0 \wedge n = 0 \wedge b = 0 \wedge l \in \mathcal{VP}\Big) \tag{A.60a}$$

$$\vee \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t = 0 \wedge n = 0 \wedge b = 0 \wedge l \in \mathcal{C} \wedge \forall l^+ \in \mathcal{C} \setminus l : knows(\overline{l^+}, 0, 0, 0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big) \tag{A.60b}$$

$$\vee \Big(\exists \mathcal{C} \in \mathcal{ISC} : \big(t = 0 \wedge n = 0 \wedge b = 0 \wedge \bar{l} \in \mathcal{C} \wedge \exists l^+ \in \mathcal{C} \setminus \bar{l} \wedge knows(l^+, 0, 0, 0) \in S_{\mathcal{D}}^{\boldsymbol{P}}\big)\Big) \tag{A.60c}$$

$$\vee \Big(\{knows(l, t-1, n, b), kNotSet(\bar{l}, t-1, n, b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n\Big) \tag{A.60d}$$

$$\vee \Big(\{knows(l, t+1, n, b), kNotSet(l, t, n, b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t < n\Big) \tag{A.60e}$$

$$\vee \Big(knows(l, t, n-1, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \tag{A.60f}$$

$$\vee \Big(kCause(l, t, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \tag{A.60g}$$

$$\vee \Big(kPosPost(l, t, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \tag{A.60h}$$

$$\vee \Big(kNegPost(l, t, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\Big) \tag{A.60i}$$

$$\vee \Big(sRes(l, n-1, b', b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t = n-1\Big) \tag{A.60j}$$

$$\vee \Big(\{sRes(l', n-1, b', b), knows(l, t, n-1, b')\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t\Big)\Big] \tag{A.60k}$$

Proof Sketch:
We investigate the domain independent theory Γ_{hpx} defined by LP rules (F1) – (F7) and the domain dependent theory Γ_{world} generated by translation rules (T1) – (T8). We identify those rules which have a predicate $knows/4$ in their head. These rules are those which define: *(a)* value propositions (T2) *(b)* initial state constraints (T3) *(c)* forward inertia (F3d) *(d)* backward inertia (F3e) *(e)* Inertia of knowledge (F3f) *(f)* causation

(F4a) *(g)* positive postdiction (F4b)*(h)* negative postdiction (F4c)*(i)* sensing (F5k) and *(j)* inheritance (F5m). The individual disjunctive elements in Equation (A.60) capture if the body of a rule with a $knows/4$-head triggers the head to be contained in the Stable Model.
This is straight forward for rules (F3d), (F3e), (F3f), (F4a),(F4b), (F4c), (F5k) and (F5m) of the domain independent theory Γ_{hpx}. Those relate to the disjunctive elements in (A.60) as follows:

- (F3d) – (A.60d)
- (F3e) – (A.60e)
- (F3f) – (A.60f)
- (F4a) – (A.60g)
- (F4b) – (A.60h)
- (F4c) – (A.60i)
- (F5k) – (A.60j)
- (F5m) – (A.60k)

LP rules generated by translation rules (T2) and (T3) relate as follows to the disjunctive elements in (A.60):

- (T2) – (A.60a)
- (T3a) – (A.60b)
- (T3b) – (A.60c)

For the "$\Leftarrow$" direction of (A.60) we argue that according to the Stable Model semantics (Gelfond and Lifschitz, 1988) the head-atom of an LP rule is contained in the Stable Model of a Logic Program if its body is "compatible" with the Stable Model, i.e. all of the rules' positive body atoms are contained in the Stable Model and all of its negative body atoms are not. Since all mentioned rules have a head atom $knows(l, t, n, b)$ the body at least one of the rules' bodies must be compatible with a Stable Model to trigger $knows(l, t, n, b)$.
For the "$\Rightarrow$" direction of (A.60) we argue similarly that according to the Stable Model semantics (Gelfond and Lifschitz, 1988), if an atom is contained in a Stable Model then there must be at least one rule in the Logic Program of which the positive body atoms are contained in the Stable Model and all of its negative body atoms are not. ■

A.4. Application of Effect Propositions

The following Lemma states that the application of effect propositions is sound. That is, whenever there exists an atom $apply(ep,t,b')$ in the Stable Model of an $\mathcal{HPX}$-Logic Program then there exists a pair $\langle ep,t\rangle$ in the Effect History $\boldsymbol{\epsilon}(\mathfrak{h})$ of a corresponding h-state.

Lemma A.7 (Soundness of application of effect propositions)

$$\begin{aligned}\forall n,b,b' :\ & hasChild(n,b,b',S^{\boldsymbol{P}}_{\mathcal{D}}) \Rightarrow \\ & \big(\forall \mathfrak{h} \in \Psi(\boldsymbol{A}_{n,b},\mathfrak{h}(n,b,S^{\boldsymbol{P}}_{\mathcal{D}})) : \\ & \qquad \forall ep,t : (apply(ep,t,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge t \leq n) \Rightarrow \langle ep,t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h})\big)\end{aligned} \tag{A.61}$$

Proof:

To prove (A.61) we make a case distinction to eliminate the $\forall b' : hasChild(n,b,b',S^{\boldsymbol{P}}_{\mathcal{D}})$-quantification. Specifically, we distinguish between (a) $\neg\exists l',b' : sRes(l',n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$ and (b) $\exists l',b' : sRes(l',n,b,b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$. Case (a) can be proven with simple substitutions and (b) requires a simple induction proof.

In both cases we argue that there are only two rules in the Logic Program with an $apply/3$ atom in the head. These are (F2a) and (F5n). Hence, if a Stable Model contains $apply/3$, then the body of one of (F2a), (F5n) must be compatible with the Stable Model. This is expressed with (A.62).

$$\begin{aligned}& \forall n,b',ep,t : \\ & apply(ep,t,b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Leftrightarrow \\ & \Big[\ \ \exists a : \Big(\{hasEP(a,ep),occ(a,t,b')\} \subset S^{\boldsymbol{P}}_{\mathcal{D}} \wedge n = t\Big) && \text{(A.62a)} \\ & \ \vee \exists b,l : \Big(\{sRes(l,n,b,b'),apply(ep,t,b)\} \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge n \geq t\Big)\Big] && \text{(A.62b)}\end{aligned}$$

Before making any case distinctions we simplify (A.61) as follows:

(A.61)

Transition function (3.7):
$\Psi(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle)$
where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,t \rangle \,| a \in \boldsymbol{A}_{n,b} \wedge t = now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\}$

$$\begin{aligned} \forall n,b,b' : \quad & hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \Big(\forall \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \\ & \quad \forall ep,t : (apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow \langle ep,t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h})\Big) \end{aligned} \tag{A.63}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,t \rangle \,| a \in \boldsymbol{A}_{n,b} \wedge t = now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\}$

According to Lemma A.13: $now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = n$

$$\begin{aligned} \forall n,b,b' : \quad & hasChild(n,b,b',S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \Big(\forall \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \\ & \quad \forall ep,t : (apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow \langle ep,t \rangle \in \boldsymbol{\epsilon}(\mathfrak{h})\Big) \end{aligned} \tag{A.64}$$

with $t \leq n$ and $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle \,| a \in \boldsymbol{A}_{n,b}\}$

(A.64)

By definition of the $eval$ function (3.17), re-evaluation does not affect the action history of an h-state. Formally:

$$\forall \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \boldsymbol{\alpha}(\mathfrak{h}) = \boldsymbol{\alpha}'$$

By the definition of effect histories (3.3) it holds that:

$$\forall \mathfrak{h} \in \bigcup_{k \in sense(\mathbf{A}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup k \rangle) : \boldsymbol{\epsilon}(\mathfrak{h}) = \boldsymbol{\epsilon}(\boldsymbol{\alpha}')$$

Hence we can eliminate the $\forall$ quantification over h-states $\mathfrak{h}$ and (A.64) is rewritten as follows:

$$\begin{aligned} \forall n, b, b' : \quad & hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \forall ep, t : (apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow \langle ep, t \rangle \in \boldsymbol{\epsilon}(\boldsymbol{\alpha}') \end{aligned} \tag{A.65}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,|a \in \boldsymbol{A}_{n,b}\}$.

By (3.3):
$\boldsymbol{\epsilon}(\boldsymbol{\alpha}') = \{\langle ep, t \rangle \,| \exists \langle a, t \rangle \in \boldsymbol{\alpha}(\mathfrak{h}') : ep \in \mathcal{EP}^a\}$

$$\begin{aligned} \forall n, b, b' : \quad & hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \forall ep, t : (apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a) \end{aligned} \tag{A.66}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Case 1 - No Sensing Results
We consider the cases where (A.67) holds. That is, the following formulae are universally quantified over those n, b for which (A.67) is true.

$$\neg\exists l', b' : sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.67}$$

We can now simplify (A.66) as follows:

(A.66)

$$\forall n, b, b' : \quad hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow$$
$$\forall ep, t : (apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a)$$

By (A.67) and (4.3):

$$((\neg\exists l' : sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \wedge hasChild(n, b, b')) \Rightarrow b = b'$$

That is, the following formulae are universally quantified over those n, b for which $\neg\exists l' : sRes(l', n, b, b')$ and $b = b'$ holds.

$$\forall ep, t : (apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a) \tag{A.68}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

There are two rules in an $\mathcal{HPX}$-Logic Program which have an atom $apply(ep, t, b)$ in the head, namely (F2a) and (F5n). We argue that $apply(ep, t, b)$ must be an atom in the Stable Model if the body of one of the rules is compatible with the Stable Model. This leads to the following case distinction:

1. Effect propositions triggered by action occurrence (F2a)

Recall (A.68):

$$\forall ep, t : (apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^{a})$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

Consider (A.62a):

$apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \exists a : (\{hasEP(a, ep), occ(a, t, b)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n = t)$

The following formulae are universally quantified over those ep for which $\exists a : (\{hasEP(a, ep), occ(a, t, b)\} \subset S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n = t)$ holds.

$$(\exists a : \{hasEP(a, ep), occ(a, n, b)\} \subset S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow (\exists \langle a, n \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^{a}) \tag{A.69}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

Since $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$ it is sufficient to show that (A.70) holds.

$$(\exists a : \{hasEP(a, ep), occ(a, n, b)\} \subset S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow (\exists a \in \boldsymbol{A}_{n,b} : ep \in \mathcal{EP}^{a}) \tag{A.70}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n \rangle \, | a \in \boldsymbol{A}_{n,b}\}$

By Definition 4.1:

$$\boldsymbol{A}_{n,b} = \{a | occ(a, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\}$$

By Lemma A.12:

$\forall a, ep : (hasEP(a, ep) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow ep \in \mathcal{EP}^{a})$

We have proven that the application of effect propositions is sound if produced by rule (F2a) (application of EP triggered by action occurrence).

2. Effect propositions triggered by inheritance (F5n)

Recall (A.68):

$$\forall ep, t : (apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t\rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a)$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

Consider (A.62b):

$$apply(ep, t, b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \exists l : (\{sRes(l, n, b, b), apply(ep, t, b)\} \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n >= t)$$

$\exists l : sRes(l, n, b, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}$ contradicts the case distinction (A.67), hence no atom $apply(ep, t, b)$ is produced.

Case 2 - With Sensing Results: We consider the cases where (A.71) holds. That is, the following formulae are universally quantified over those n, b for which (A.71) is true.

$$\exists l' : sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \tag{A.71}$$

The case distinction allows us to simplify (A.66). Consider the following substitutions:

Recall (A.66):

$$\begin{aligned}\forall n, b, b' : \quad & hasChild(n, b, b', S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow \\ & \forall ep, t : (apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t\rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a)\end{aligned}$$

By (A.71) and (4.3):

$$\forall b' : (\exists l' : sRes(l', n, b, b')) \Rightarrow hasChild(n, b, b') = true$$

The following formulae are universally quantified over those n, b for which (A.71) is true.

$$\forall ep, t, b' : (apply(ep, t, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a, t\rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a) \tag{A.72}$$

where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a, n\rangle \,|a \in \boldsymbol{A}_{n,b}\}$

To prove that (A.72) holds, we consider both rules of the $\mathcal{HPX}$-LP with an atom $apply(ep, t, b')$ in the head: (F2a) and (F5n). We argue that $apply(ep, t, b')$ must be an atom in the Stable Model if the body of one of the rules is compatible with the Stable Model.

To this end, we perform induction over the structure of (A.62) for b'. For the base step show that (A.72) holds for those b' for which an atom $apply(ep, t, b')$ produced by rule (F2a). For the induction step we consider rule (F5n) which involves another

$apply(ep,t,b'')$ atom: we argue that if (A.72) holds for a b' and if an atom $apply(ep,t,b'')$ is produced by rule (F5n), then (A.72) also holds for the b''. The induction is complete because rules (F2a) and (F5n) are the only rules with $apply(ep,t,b')$ atoms in the head.

1. Base Step: effect propositions triggered by action occurrence (F2a)
   Consider (A.62a):

$$apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \exists a : \{hasEP(a,ep), occ(a,t,b')\} \subset S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t = n \quad \text{(A.73)}$$

   Due to Lemma A.9 it holds that $\neg\exists b', l' : (b' \neq b \wedge sRes(l',n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge occ(a,n,b') \in S_{\mathcal{D}}^{\boldsymbol{P}})$. Hence, we may only consider cases where $b' = b$. In this case, the proof is analogous to the soundness proof for effect propositions triggered by action occurrence in Case 1.

2. Induction Step: effect propositions triggered by inheritance (F5n)

   Recall (A.68):

$$\forall ep,t : (apply(ep,t,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a,t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a)$$

   where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle | a \in \boldsymbol{A}_{n,b}\}$

   Consider (A.62a):

   $apply(ep,t,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftarrow \exists l : (\{sRes(l,n,b,b'), apply(ep,t,b)\} \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n >= t)$

   The following formulae are universally quantified for those ep for which $\exists l : (\{sRes(l,n,b,b'), apply(ep,t,b)\} \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n >= t)$ holds.

$$(\exists l : sRes(l,n,b,b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge apply(ep,t,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge n \geq t) \Rightarrow (\exists \langle a,t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a) \quad \text{(A.74)}$$

   where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \cup \{\langle a,n \rangle | a \in \boldsymbol{A}_{n,b}\}$

   Since we perform induction we assume that soundness holds for $apply(ep,t,b)$. That is,

   $(apply(ep,t,b) \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge t \leq n) \Rightarrow (\exists \langle a,t \rangle \in \boldsymbol{\alpha}' : ep \in \mathcal{EP}^a)$ (A.75)

   We have shown that (A.72) holds for $apply/3$ produced by the inheritance rule (F5n).

We have shown that the Lemma holds by proving that (A.61) holds for both cases, $\neg\exists b', l' : sRes(l', n, b, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$ and $\exists b', l' : sRes(l', n, b, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$. ■

Lemma A.8 (Branching of application of effect propositions)

$$\begin{aligned} &\forall l, n, b, b', ep, t : \\ &(\{sRes(l', n, b, b'), apply(ep, t, b)\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}} \wedge t \leq n) \quad \Leftrightarrow \\ &(\{sRes(l', n, b, b'), apply(ep, t, b')\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}} \wedge t \leq n) \end{aligned} \tag{A.76}$$

Proof:
We distinguish two cases: The $\Rightarrow$ direction directly emerges from the inheritance rule (F5n). For the $\Leftarrow$ direction we consider two cases:

1. Consider that an atom $apply(ep, t, b')$ is produced by rule (F2a). In this case it must hold that $\{occ(a, n, b'), hasEP(a, ep)\} \subseteq S^{\boldsymbol{P}}_{\mathcal{D}} \wedge n = t$ and by Definition 4.1 it holds that $occ(a, n, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \Rightarrow uBr(n, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$. However, considering that $sRes(l', n, b, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$ the integrity constraint (F5i) assures that $uBr(n, b') \notin S^{\boldsymbol{P}}_{\mathcal{D}}$ and leads to a contradiction. Hence $apply(ep, t, b')$ can not be produced by (F2a) if $sRes(l', n, b, b')S^{\boldsymbol{P}}_{\mathcal{D}}$

2. Consider that an atom $apply(ep, t, b')$ is produced by rule (F5n). Then clearly it must hold that $apply(ep, t, b) \in S^{\boldsymbol{P}}_{\mathcal{D}}$

■

Lemma A.9 (Actions do not occur in new branches) *:*

$$\begin{aligned} &\forall n, b, a : \\ &\neg\exists b', l' : (b' \neq b \wedge sRes(l', n, b, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge occ(a, n, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}) \end{aligned} \tag{A.77}$$

Proof: Suppose the contrary is true, i.e. $\exists b', l' : (b' \neq b \wedge sRes(l', n, b, b') \in S^{\boldsymbol{P}}_{\mathcal{D}} \wedge occ(a, n, b') \in S^{\boldsymbol{P}}_{\mathcal{D}})$. Then by definition 4.1 if must hold that $uBr(n, b') \in S^{\boldsymbol{P}}_{\mathcal{D}}$. This again contradicts the integrity constraint (F5i), hence (A.77) must hold. ■

A.5. Sensing Results

We prove soundness for sensing results: if at a node n, b an atom $sRes(l, n, b, b')$ is produced, then the $sense$-function (3.8) returns a pair $\langle l, t\rangle$. This is formally expressed with Lemma A.10:

Lemma A.10 (Soundness for sensing results)

$$\forall l, n, b, b' : \\ sRes(l, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\langle l, n\rangle, \langle \bar{l}, n\rangle\} \tag{A.78}$$

$$\forall n, b, b' : \\ (\neg \exists l : sRes(l, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow (sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\emptyset\}) \tag{A.79}$$

Proof:
Consider all rules in an $\mathcal{HPX}$-Logic Program with an $sRes/4$ atom in the head. These are:

$$sRes(F, N, B, B) \leftarrow occ(A, N, B), hasKP(A, F), \\ not\ kw(F, N, N, B) \tag{F5f}$$

$$1\{sRes(neg(F), N, B, B') : neq(B, B')\}1 \leftarrow occ(A, N, B), hasKP(A, F), \\ not\ kw(F, N, N, B) \tag{F5g}$$

We need the following auxiliary result. Consider rules (F5d),(F5e) which produce $kw/4$ atoms. According to the Stable Model semantics, since these rules are the only rules with $kw/4$ in their heads the following must hold:

$$\forall f, t, n, b : \\ kw(f, t, n, b) \notin S_{\mathcal{D}}^{\boldsymbol{P}} \Rightarrow \{knows(f, t, n, b), knows(neg(f), t, n, b)\} \cap S_{\mathcal{D}}^{\boldsymbol{P}} = \emptyset \tag{A.80}$$

To prove that (A.78) holds, we show that for both rules (F5f), (F5g) that if their body is compatible with the Stable Model $S_{\mathcal{D}}^{\boldsymbol{P}}$, then $\exists \langle l, n\rangle \in sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}))$ must hold. That is, to show that (A.78) holds we consider (A.80) and show that both (A.81) and (A.82) hold:

$$\forall f, n, b : \\ \exists a : \big(\{occ(a, n, b), hasKP(a, f)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ \{knows(f, n, n, b), knows(neg(f), n, n, b)\} \cap S_{\mathcal{D}}^{\boldsymbol{P}} = \emptyset\big) \\ \Rightarrow \\ (sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\langle l, n\rangle, \langle \bar{l}, n\rangle\}) \tag{A.81}$$

$$\forall f, n, b : \\ \exists a : \big(\{occ(a, n, b), hasKP(a, f)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}} \wedge \\ \{knows(f, n, n, b), knows(neg(f), n, n, b)\} \cap S_{\mathcal{D}}^{\boldsymbol{P}} = \emptyset\big) \\ \Rightarrow \\ (sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\langle l, n\rangle, \langle \bar{l}, n\rangle\}) \tag{A.82}$$

1. Positive sensing result (A.81)

(A.81)

With Definition 4.1 and Lemma A.12:

$$\exists a : \big(\{occ(a,n,b), hasKP(a,f)\} \subseteq S_{\mathcal{D}}^{\boldsymbol{P}}\big) \\ \Rightarrow (\exists a \in \boldsymbol{A}_{n,b} : \mathcal{KP}^a = f)$$

$$\begin{aligned} &\forall f,n,b : \\ &\exists a \in \boldsymbol{A}_{n,b} : \big(\mathcal{KP}^a = f \wedge \\ &\qquad \{knows(f,n,n,b), knows(neg(f),n,n,b)\} \cap S_{\mathcal{D}}^{\boldsymbol{P}} = \emptyset\big) \\ &\Rightarrow (sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\langle f,n\rangle, \langle \neg f,n\rangle\}) \end{aligned} \tag{A.83}$$

By (4.5): $\boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) = \{\langle l,t\rangle \,|knows(l,t,n,b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\}$

$$\{knows(f,n,n,b), knows(neg(f),n,n,b)\} \cap S_{\mathcal{D}}^{\boldsymbol{P}} = \emptyset \\ \Rightarrow \{\langle \neg f,n\rangle, \langle f,n\rangle\} \cap \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) = \emptyset$$

$$\begin{aligned} &\forall f,n,b : \\ &\exists a \in \boldsymbol{A}_{n,b} : \mathcal{KP}^a = f \wedge \\ &\qquad \{\langle \neg f,n\rangle, \langle f,n\rangle\} \cap \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) = \emptyset\big) \\ &\Rightarrow (sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \{\langle f,n\rangle, \langle \neg f,n\rangle\}) \end{aligned} \tag{A.84}$$

(A.84)

Consider (3.8):

$$sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \begin{cases} \{\{\langle f, now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\rangle\}, \{\langle \neg f, now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\rangle\}\} \\ \quad \text{if } \exists a \in \boldsymbol{A}_{n,b} : \mathcal{KP}^a = f \wedge \\ \qquad \langle f, now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\rangle \notin \boldsymbol{\kappa}(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) \wedge \\ \qquad \langle \neg f, now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}))\rangle \notin \boldsymbol{\kappa}(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) \\ \{\emptyset\} \quad \text{otherwise} \end{cases}$$

With (4.5): $\boldsymbol{\kappa}(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})$
According to now (3.5), (4.3) and (4.5): $now(\mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = n$

$$sense(\boldsymbol{A}_{n,b}, \mathfrak{h}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}})) = \begin{cases} \{\{\langle f, n\rangle\}, \{\langle \neg f, n\rangle\}\} \\ \quad \text{if } \exists a \in \boldsymbol{A}_{n,b} : \mathcal{KP}^a = f \wedge \\ \qquad \langle f, n\rangle \notin \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \wedge \\ \qquad \langle \neg f, n\rangle \notin \boldsymbol{\kappa}(n,b,S_{\mathcal{D}}^{\boldsymbol{P}}) \\ \{\emptyset\} \quad \text{otherwise} \end{cases}$$

It follows that (A.84) holds.

2. Negative sensing result (A.82)
   This is analogous to the case of the positive sensing result.

The proof for (A.79) is analogous to the proof for (A.78). ■

Lemma A.11 (Only one sensing result per branch)

$$\forall n, l, b, b', l' : ((sRes(l, n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}} \wedge sRes(l', n, b, b') \in S_{\mathcal{D}}^{\boldsymbol{P}}) \Rightarrow l = l') \quad \text{(A.85)}$$

Proof Sketch:
This directly follows from the integrity constraint (F5h). ■

A.6. Auxiliary Lemmata

Soundness of auxiliary predicates

The following lemma concerns soundness of auxiliary predicates in the ASP formalization of $\mathcal{HPX}$.

Lemma A.12 (Soundness for auxiliary predicates) *Given the prerequisites described in Definition 4.1 the following holds:*

1. $hasEP(a, ep) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow ep \in \mathcal{EP}^a$
2. $hasEff(ep, l) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow e(ep) = l$
3. $hasCond(ep, l) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow l \in c(ep)$
4. $hasKP(a, f) \in S_{\mathcal{D}}^{\boldsymbol{P}} \Leftrightarrow \mathcal{KP}^a = f$

Proof:
This follows directly from observing that the auxiliary predicates $hasEP/2$, $hasEff/2$, $hasCond/2$, $hasKP/2$ only appear as facts in the Logic Program $LP(\mathcal{D}) \cup \boldsymbol{P}$ if they are produced by translation rules (T5) – (T7). There are no other rules which have one of the auxiliary predicates in their head. ■

Current Step Number

The following lemma states, that if an h-state is extracted from a Stable Model (denoted $\mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})$, see (4.5)), then the step number of that state (denoted by $now(\mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}})$, see (3.5)) equals n.

Lemma A.13 (Current Step Number) *Given the prerequisites described in Definition 4.1 the following holds:*

$$now(\mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) = n \tag{A.86}$$

Proof Sketch:
Consider (4.5):

$$\mathfrak{h}(n, b, S) = \langle \boldsymbol{\alpha}(n, b, S), \boldsymbol{\kappa}(n, b, S) \rangle$$
$$\boldsymbol{\alpha}(n, b, S) = \{\langle a, t \rangle \,|\exists b', t : (occ(a, t, b') \in S \wedge ancestor(t, b', n, b, S))\}$$

It follows from the definition of $anchestor$ (4.4) that $\forall \langle a, t \rangle \in \boldsymbol{\alpha}(n, b, S) : t < n$. It follows from the Definition 4.1 that there are no gaps in the plan, i.e. $\forall n, b : (uBr(n, b) \Rightarrow \exists a : occ(a, n, b))$ Then, by the definition of now (3.5) we have $now(\mathfrak{h}(n, b, S_{\mathcal{D}}^{\boldsymbol{P}}) = n$.

■

$\mathcal{B}$

Computational Properties of $\mathcal{HPX}$

This appendix contains proofs concerning the computational complexity and other properties of $\mathcal{HPX}$. As argued in Section 3.3 we assume that the size of concurrent conditional plans (CCP) is of polynomial size wrt. $\mathcal{D}$.

B.1. Computational Complexity

We prove Theorem 3.1 pertaining the computational worst-time complexity for $\mathcal{HPX}$: *Let $\mathcal{D}$ be a planning domain, then deciding whether*

$$\exists p : solves(p, \mathcal{D}) \tag{B.1}$$

holds is in NP.

Proof: The problem of deciding whether an expression of the form $\exists u P(u, w)$ is true or false is in NP if (1) u runs over words of polynomial length and (2) checking whether $P(u, w)$ holds is polynomial.
Consider that $u = p$ is a concurrent conditional plan and $w = \mathcal{D}$ is a domain specification such that $P(u, w) = solves(p, \mathcal{D})$. (1) is given by the restriction that only plans of polynomial size are considered (see argumentation in Section 3.3). To show that (2) also holds, we prove that determining whether a plan p solves a planning domain $\mathcal{D}$ is polynomial in Lemma B.1. This proves Theorem 3.1. ■

Lemma B.1 (Solving the projection problem is polynomial) *Given a concurrent conditional plan (CCP) p and a domain $\mathcal{D}$. Deciding whether*

$$solves(p, \mathcal{D}) \tag{B.2}$$

holds is polynomial.

Proof: Reconsider function $solves(p, \mathcal{D})$:

$$\begin{aligned} solves(p, \mathcal{D}) = \quad & \forall \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : \forall l^{sg} \in \mathcal{G}^{strong} : \mathfrak{h} \models l^{sg} \\ & \wedge \exists \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : \forall l^{wg} \in \mathcal{G}^{weak} : \mathfrak{h} \models l^{wg} \end{aligned}$$

To determine whether a plan p solves a problem domain $\mathcal{D}$, we check whether strong goals holds in all leafs and weak goals hold in at least one leaf of the transition tree. Checking whether a goal holds in all leafs of the transition tree is linear wrt. the number of leafs and goal literals. It follows from the transition function (3.7) and the extended transition function (3.18) that the number of leafs is less or equal to the number of sensing actions in a plan p. The number of sensing actions and hence the number of leafs is polynomial due to the restriction that we only consider plans of polynomial size. As of Lemma B.2, applying the extended transition function (3.18) is polynomial. Thus, to solve the projection problem we apply a polynomial number of polynomial operations which is again polynomial. ∎

Lemma B.2 (Applying the extended transition function is polynomial) *Given a plan p and a consistent h-state $\mathfrak{h}$, applying the extended transition function $\widehat{\Psi}(p, \mathfrak{h}_0)$ (3.18) is polynomial.*

Proof: In Lemma B.3 we show that applying the transition function (3.7) for a set of actions is polynomial. The extended transition (3.18) applies the transition function (3.7) once for each concurrent set of actions in the plan. As plans are restricted to be of polynomial size, the transition function is applied polynomially often. Consequently, with the extended transition function we perform a polynomial number of polynomial operations, which is again polynomial. ∎

Lemma B.3 (Applying the transition function is polynomial) *Given a set of actions $\boldsymbol{A}$ and a consistent h-state $\mathfrak{h}$, applying the transition function $\Psi(\boldsymbol{A}, \mathfrak{h})$ (3.7) is polynomial.*

Proof: Recall the transition function (3.7):

$$\Psi(\mathbf{A}, \mathfrak{h}) = \bigcup_{k \in sense(\mathbf{A}^{ex}, \mathfrak{h})} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k \rangle)$$

The transition function calls the $eval$ function (3.17) and conjoins its result with sensing results. Conjoining the sensing results is done in constant time and the number of potential sensing results is $|k| \leq 2$. Therefore the computational worst-time complexity is determined by $eval$.
To see that $eval$ is polynomial, consider the following: $eval(\mathfrak{h})$ calls $evalOnce(\mathfrak{h})$ until $\mathfrak{h}$ converged. We must therefore show that $evalOnce(\mathfrak{h})$ is (i) itself polynomial, and (ii) called at most polynomially often wrt. $\mathfrak{h}$. (i) is shown with Lemma (B.4) and (ii) is shown with Lemma (B.5). ∎

Lemma B.4 (Applying $evalOnce$ (B.2) is polynomial) *Applying $evalOnce(\mathfrak{h})$ (B.2) is polynomial for an arbitrary h-state $\mathfrak{h}$ and a domain $\mathcal{D}$.*

Proof: $evalOnce$ (B.2) calls the five inference mechanisms (3.11) – (3.15), i.e. fwd, $back$, $causal$, pd^{pos} and pd^{neg}. All atomic operations (like concatenation of sets) in (3.11) – (3.15) are executed in linear or constant time. Quantifications in (3.11) – (3.15) are always either over the applied effect propositions, the condition literals in an effect proposition or over the elements in $\boldsymbol{\kappa}(\mathfrak{h})$ which are all sets of constant size wrt. $\mathfrak{h}$ and $\mathcal{D}$. Therefore $evalOnce(\mathfrak{h})$ is polynomial wrt. $\mathfrak{h}$ and $\mathcal{D}$. ■

Lemma B.5 (Function $evalOnce$ is called constantly often by $eval$) *Let $|\mathcal{L}_\mathcal{D}|$ be the number of literals in a domain $\mathcal{D}$ with the initial h-state $\mathfrak{h}_0$. Let p be a conditional plan and $\mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0)$ be an arbitrary leaf state resulting from applying the extended transition function. Then $evalOnce(\mathfrak{h})$ (B.2) is called at most $|\mathcal{L}_\mathcal{D}| \cdot now(\mathfrak{h})$ times by $eval(\mathfrak{h})$ (3.17).*

Proof: As of Lemma B.6 it holds for all h-states $\mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0)$ that the maximum size of the knowledge history $\boldsymbol{\kappa}(\mathfrak{h})$ is $|\mathcal{L}_\mathcal{D}| \cdot now(\mathfrak{h})$. Lemma B.7 proves monotonicity of $evalOnce$: that is, for a pair $\langle l, t\rangle$, if $\mathfrak{h} \models \langle l, t\rangle$ then $evalOnce(\mathfrak{h}) \models \langle l, t\rangle$. Therefore at most $|\mathcal{L}_\mathcal{D}| \cdot now(\mathfrak{h})$ changes can be made to $\boldsymbol{\kappa}(\mathfrak{h})$ until convergence is achieved. Thus, $evalOnce$ can only be called at most $|\mathcal{L}_\mathcal{D}| \cdot now(\mathfrak{h})$ times. ■

Lemma B.6 (Maximal size of knowledge history) *Let $\mathcal{D}$ be a domain with the initial h-state $\mathfrak{h}_0$ and p be a concurrent conditional plan. Let $|\mathcal{L}_\mathcal{D}|$ be the number of literals in $\mathcal{D}$. Then the following holds for the number of elements in the knowledge history:*

$$\forall \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : |\boldsymbol{\kappa}(\mathfrak{h})| \leq |\mathcal{L}_\mathcal{D}| \cdot now(\mathfrak{h}) \tag{B.3}$$

Proof Sketch: The knowledge history $\boldsymbol{\kappa}(\mathfrak{h})$ consists of pairs $\langle l, t\rangle$, where $l \in \mathcal{L}_\mathcal{D}$. Since $\mathcal{L}_\mathcal{D}$ is determined by the domain size it is sufficient to show that for all t it holds that $0 \leq t \leq now(\mathfrak{h})$. Formally:

$$\forall \mathfrak{h} \in \widehat{\Psi}(p, \mathfrak{h}_0) : \forall \langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) : 0 \leq t \leq now(\mathfrak{h}) \tag{B.4}$$

To prove (B.4) we prove the following more general proposition (B.5).

$$\forall \mathfrak{h}' \in \widehat{\Psi}(p, \mathfrak{h}) : \forall \langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}') : 0 \leq t \leq now(\mathfrak{h}') \tag{B.5}$$

where $\mathfrak{h}$ is an arbitrary h-state such that $0 \leq t \leq now(\mathfrak{h})$. This generalization is valid because by Definition 3.2 for any initial h-state $\mathfrak{h}_0$ it holds that $0 \leq t \leq now(\mathfrak{h}_0)$.
To prove (B.4) we perform nested induction. The "outer" induction is over the structure of a concurrent conditional plan p. Most steps are trivial, except for the second base step where an "inner" induction is required. The inner induction is over the structure of knowledge producing mechanisms which we describe in Lemma B.9.

- Outer base step 1: $p = []$
  This case clearly holds because by the extended transition function (3.18) it holds that $\widehat{\Psi}([], \mathfrak{h}) = \{\mathfrak{h}\}$.

- Outer base step 2: $p = [a_1||\cdots||a_n]$
  In this case we have according to the extended transition function (3.18):

$$\widehat{\Psi}([a_1||\cdots||a_n], \mathfrak{h}) = \Psi([a_1||\cdots||a_n], \mathfrak{h})$$

  We consider Lemma B.9 which identifies all mechanisms within $\Psi([a_1||\cdots||a_n], \mathfrak{h})$ which produce pairs $\langle l, t\rangle$ and prove inductively that (B.5) holds:

  - Inner base step 1: $\langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h})$
    This emerges directly from the premise that $0 \leq t \leq now(\mathfrak{h})$.
  - Inner base step 2:$\{\langle l, t\rangle\} \in sense(\{\{a_1, \ldots, a_n\}, \mathfrak{h}\})$
    By definition of $sense$ (3.8) it holds that $t = now(\mathfrak{h})$.
  - Inner base step 3:$\langle l, t\rangle \in add_{cause}(\mathfrak{h}')$
    Consider (3.13): $add_{cause}(\mathfrak{h}')$ can not produce a pair $\langle l, t\rangle$ with $t < 1$ or $t > now(\mathfrak{h}')$ because by the Definition 3.1 of the effect history $\boldsymbol{\epsilon}(\mathfrak{h}')$ and the transition function (3.7) we know that $\forall\, \langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}') : t < now(\mathfrak{h}') \wedge t \geq 0$.
  - Inner base step 4,5:$\langle l, t\rangle \in add_{pd^{pos}}(\mathfrak{h}')$ or $\langle l, t\rangle \in add_{pd^{neg}}(\mathfrak{h}')$.
    These cases are similar to $add_{cause}(\mathfrak{h}')$.
  - Inner induction step 1: $\langle l, t\rangle \in add_{fwd}(\mathfrak{h}')$.
    Reconsider (3.11):

$$add_{fwd}(\mathfrak{h}') = \{\langle l, t\rangle \,|\, \langle l, t-1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}') \wedge inertial(l, t{-}1, \mathfrak{h}') \wedge t \leq now(\mathfrak{h}')\}$$

    It holds that $\forall t : t \leq now(\mathfrak{h}')$. By inner induction hypothesis we assume that $t - 1 \geq 0$ and therefore it must hold that $t > 0$.
  - Inner induction step 2: $\langle l, t\rangle \in add_{back}(\mathfrak{h}')$.
    This is analogous to inner induction step 1.

- Outer induction step 1: $p = [p_1; p_2]$
  It holds that $\widehat{\Psi}([p_1; p_2], \mathfrak{h}) = \bigcup_{\mathfrak{h}_i \in \widehat{\Psi}(p_1, \mathfrak{h})} \widehat{\Psi}(p_2, \mathfrak{h}_i)$ By outer hypothesis we assume that *(a)* $\forall \mathfrak{h}_i \in \widehat{\Psi}(p_1, \mathfrak{h}) : 0 \leq t \leq now(\mathfrak{h}_i)$ and *(b)* $\forall \mathfrak{h}_i \in \widehat{\Psi}(p_1, \mathfrak{h}) : \forall \mathfrak{h}' \in \widehat{\Psi}(p_2, \mathfrak{h}_i) : 0 \leq t \leq now(\mathfrak{h}')$.

- Outer induction step 2: $p =$`if` l `then` p_1 `else` p_2 This is similar to outer induction step 1.

We have shown by induction that (B.5) holds. (B.5) is a generalization of (B.4). ∎

B.2. Knowledge-persistence and Monotonicity of Re-evaluation

The following Lemmata capture that knowledge can not get lost, i.e. $\mathcal{HPX}$-agents do not “forget” knowledge.

Lemma B.7 (Re-evaluation is monotonic) *Let $\mathcal{D}$ be a domain description with an initial state $\mathfrak{h}_0$ and p be a conditional concurrent plan. For all leaf states $\mathfrak{h} \in \widehat{\Psi(p, \mathfrak{h}_0)}$ it holds that*

$$\forall \langle l,t\rangle : (\mathfrak{h} \models \langle l,t\rangle \Rightarrow evalOnce(\mathfrak{h}) \models \langle l,t\rangle) \tag{B.6}$$

and

$$\forall \langle l,t\rangle : (\mathfrak{h} \models \langle l,t\rangle \Rightarrow eval(\mathfrak{h}) \models \langle l,t\rangle) \tag{B.7}$$

Proof: The proof of (B.6) follows from a simple syntactic investigation of five IM (3.11) – (3.15) which are called by $evalOnce$. Recall the definition of $evalOnce$ (B.2):

$$evalOnce(\mathfrak{h}) = pd^{neg}(pd^{pos}(cause(back(fwd(\mathfrak{h})))))$$

Let the following hold for an h-state $\mathfrak{h} = \langle \boldsymbol{\alpha}, \boldsymbol{\kappa}\rangle$:

$$\begin{aligned}
fwd(\mathfrak{h}) &= \tilde{\mathfrak{h}}_{fwd} = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{fwd}(\mathfrak{h})\rangle \\
back(fwd(\mathfrak{h})) &= \tilde{\mathfrak{h}}_{back} = \left\langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}_{fwd}) \cup add_{back}(\mathfrak{h}_{fwd})\right\rangle \\
cause(back(fwd(\mathfrak{h}))) &= \tilde{\mathfrak{h}}_{cause} = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}_{back}) \cup add_{cause}(\mathfrak{h}_{back})\rangle \\
pd^{pos}(cause(back(fwd(\mathfrak{h})))) &= \tilde{\mathfrak{h}}_{pd^{pos}} = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}_{cause}) \cup add_{pd^{pos}}(\mathfrak{h}_{cause})\rangle \\
pd^{neg}(pd^{pos}(cause(back(fwd(\mathfrak{h}))))) &= \tilde{\mathfrak{h}}_{pd^{neg}} = \left\langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}_{pd^{pos}}) \cup add_{pd^{neg}}(\mathfrak{h}_{pd^{pos}})\right\rangle
\end{aligned} \tag{B.8}$$

then $evalOnce(\mathfrak{h}) = \tilde{\mathfrak{h}}_{pd^{neg}}$.
From (B.8) we extract the following implications:

$$\begin{aligned}
\forall \langle l,t\rangle : \big(& \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{fwd}) \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{back}) \\
&\Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{cause}) \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{pd^{pos}}) \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(evalOnce(\mathfrak{h}))\big)
\end{aligned} \tag{B.9}$$

It follows by (B.9) that

$$\forall \langle l,t\rangle : (\langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \Rightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(evalOnce(\mathfrak{h}))) \tag{B.10}$$

The definition of the $\models$ operator (3.6b) is :

$$\forall l,t : (\mathfrak{h} \models \langle l,t\rangle \Leftrightarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h})) \tag{B.11}$$

Consequently, $\forall \langle l,t\rangle : \mathfrak{h} \models \langle l,t\rangle \Rightarrow evalOnce(\mathfrak{h}) \models \langle l,t\rangle$ (B.6) is true.
The proof of (B.7) follows from (B.6) and the recursive definition of $eval$ (3.17). ■

Lemma B.8 (Knowledge-persistence for CCP) *Consider an h-state $\mathfrak{h}$ a concurrent conditional plan p. Then (B.12) holds:*

$$\forall l, t, \mathfrak{h}' \in \widehat{\Psi}(p, \mathfrak{h}) : \big(\mathfrak{h} \models \langle l, t\rangle \Rightarrow \mathfrak{h}' \models \langle l, t\rangle\big) \tag{B.12}$$

Proof: We perform induction over the structure of a CCP p.

1. $p = []$
   In this case $\widehat{\Psi}([], \mathfrak{h}) = \{\mathfrak{h}\}$ and the lemma trivially holds.

2. $p = [a_1||\ldots||a_n]$
   In this case $\widehat{\Psi}([a_1||\ldots||a_n], \mathfrak{h}) = \Psi(\{a_1, \ldots, a_n\}, \mathfrak{h})$. Recall the transition function (3.7):

$$\Psi(\{a_1, \ldots, a_n\}, \mathfrak{h}) = \bigcup_{k \in sense(\{a_1,\ldots,a_n\},\mathfrak{h})} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle) \tag{B.13}$$

   If follows trivially that (B.14) holds.

$$\forall l, t, k \in sense(\{a_1, \ldots, a_n\}, \mathfrak{h}) : \big(\mathfrak{h} \models \langle l, t\rangle \Rightarrow \boldsymbol{\kappa}(\mathfrak{h}) \cup k \models \langle l, t\rangle\big) \tag{B.14}$$

   Lemma B.7 shows that $eval$ is monotonic, i.e. (B.15) holds.

$$\forall l, t, k \in sense(\{a_1, \ldots, a_n\}, \mathfrak{h}) : \big(\mathfrak{h} \models \langle l, t\rangle \Rightarrow eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle) \models \langle l, t\rangle\big) \tag{B.15}$$

   Consequently, (B.13) is true.

3. $p = [p_1; p_2]$ where p_1, p_2 are CCP
   This follows directly from the induction hypothesis and the extended transition function (3.18).

4. $p = [\texttt{if}\ l\ \texttt{then}\ p_1\ \texttt{else}\ p_2]$ where p_1, p_2 are CCP
   This follows directly from the induction hypothesis and the extended transition function (3.18).

■

B.3. Knowledge Producing Mechanisms

The following Lemma states that if knowledge is produced then it is either produced by sensing or one of the five inference mechanisms:

Lemma B.9 (Knowledge producing mechanisms for single state transitions) *Given a domain $\mathcal{D}$, an h-state $\mathfrak{h}$ and a set of actions $\boldsymbol{A}$. Then for all h-states $\mathfrak{h}' \in \Psi(\boldsymbol{A}, \mathfrak{h})$ it holds that a pair $\langle l, t\rangle$ can only be contained in the knowledge history $\boldsymbol{\kappa}(\mathfrak{h}')$ if and only if it was produced by sensing or one of the inference mechanisms* **IM.1–IM.5** *(3.11) – (3.15). This is formally expressed as follows:*

$$\begin{aligned} \forall \langle l, t\rangle : \langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}') \Leftrightarrow \\ \Big(& \langle l, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}) \\ & \vee \langle l, t\rangle \in add_{fwd}(\mathfrak{h}') \\ & \vee \langle l, t\rangle \in add_{back}(\mathfrak{h}') \\ & \vee \langle l, t\rangle \in add_{cause}(\mathfrak{h}') \\ & \vee \langle l, t\rangle \in add_{pd^{pos}}(\mathfrak{h}') \\ & \vee \langle l, t\rangle \in add_{pd^{neg}}(\mathfrak{h}') \\ & \vee \{\langle l, t\rangle\} \in sense(\{\boldsymbol{A}, \mathfrak{h}\})\Big) \end{aligned} \tag{B.16}$$

Proof:
This follows from the constitution of the transition function (3.7) and the re-evaluation functions. Recall (3.7):

$$\Psi(\boldsymbol{A}, \mathfrak{h}) = \bigcup_{k \in sense(\boldsymbol{A}, \mathfrak{h})} eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle)$$

where $eval$ is recursively defined as follows:

$$eval(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle) = \begin{cases} \mathfrak{h} & \text{if } evalOnce(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle) = \langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle \\ eval(evalOnce(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle)) & \text{otherwise} \end{cases}$$

Recall $evalOnce$ (refeq:evalOnce):

$$evalOnce(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle) = pd^{neg}(pd^{pos}(cause(back(fwd(\langle \boldsymbol{\alpha}', \boldsymbol{\kappa}(\mathfrak{h}) \cup k\rangle)))))$$

The $\Rightarrow$ direction of (B.16) follows directly from syntactic investigation of Ψ, $eval$ and the constitution of the inference mechanism functions fwd, $back$, $cause$, etc.: each of the five inference mechanisms calls one add-function (add_{fwd}, add_{back}, add_{cause}, etc.) which generates additional pairs $\langle l, t\rangle$. For example $fwd(\mathfrak{h}) = \langle \boldsymbol{\alpha}(\mathfrak{h}), \boldsymbol{\kappa}(\mathfrak{h}) \cup add_{fwd}(\mathfrak{h})\rangle$.
By Lemma B.7 it holds that no knowledge is removed from the knowledge history of an h-state. Hence if for some h-state $\mathfrak{h}$ it holds that $\langle l, t\rangle \in add_{\boldsymbol{IM}}(\mathfrak{h})$, then $\langle l, t\rangle \in eval(\mathfrak{h})$ (where $\boldsymbol{IM}$ is either add,$back$,$cause$, etc.). This proves the $\Rightarrow$ direction of (B.16).
The $\Leftarrow$ direction of (B.16) follows from the observation that there is no other operation within the transition function which generates a pair $\langle l, t\rangle$. ∎

C

Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$

We prove Theorem 3.3 which states that $\mathcal{HPX}$ is sound wrt. the ${\mathcal{A}_k}^{TQS}$ semantics. Due to restrictions in $\mathcal{A}_k$, we forbid that actions can happen concurrently. That is, if an action has a knowledge proposition then it can not have an effect proposition.
The proof is done by induction over the number of actions. In the remainder of the proof we use the following notational conventions:

- $\mathcal{D}$ is a domain specification.
- $\alpha_n = [a_1; \ldots; a_n]$ and $\alpha_{n+1} = [a_1; \ldots; a_n; a_{n+1}]$ are sequences of actions.
- $\mathfrak{h}_0$ is the initial h-state of $\mathcal{D}$.
- $\mathfrak{h}_n \in \widehat{\Psi}([a_1; \ldots; a_n], \mathfrak{h}_0)$ is an h-state which results from applying the extended transition function on $\mathfrak{h}_0$. Similar for $\mathfrak{h}_{n+1}$.
- $\delta_0 = \langle u_0, \Sigma_0 \rangle$ is an arbitrary valid initial c-state of $\mathcal{D}$.
- $\delta_n = \langle u_n, \Sigma_n \rangle = \Phi(a_n, \Phi(a_{n-1}, \cdots \Phi(a_1, \delta_0)))$ is a c-state obtained by applying the $\mathcal{A}_k$ transition functions (3.24), (3.26). Similar for δ_{n+1}.
- Σ_n^0 and Σ_{n+1}^0 are the re-evaluated initial k-states as described in Definition 3.6:
$$\Sigma_n^0 = \{s_0 | s_0 \in \Sigma_0 \wedge Res(a_n, Res(a_{n-1}, \cdots Res(a_1, s_0))) \in \Sigma_n\}$$
and similar for Σ_{n+1}^0.
- Σ_n^t and Σ_{n+1}^t are re-evaluated k-states as described in Definition 3.7:
$$\Sigma_n^t = \bigcup_{s \in \Sigma_n^0} Res(a_t, Res(a_{t-1}, \cdots Res(a_1, s)))$$
with $0 \leq t \leq n$ and similar for Σ_{n+1}^t with $0 \leq t \leq n+1$.

With the conventions Theorem 3.3 is rewritten as Lemma C.1.

Lemma C.1 (Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$ for sequences of actions)

$$\begin{aligned}&\forall n : \exists \mathfrak{h}_n \in \widehat{\Psi}(\alpha_n, \mathfrak{h}_0) : \forall l, t : \\ &\mathfrak{h}_n \models \langle l, t\rangle \Rightarrow \Sigma_n^t \models l \\ &\textit{with } t \leq n.\end{aligned} \tag{C.1}$$

Proof: Induction over the number of actions n. The base step $n = 0$ is stated in Lemma C.2. The induction step $(n \to n + 1)$ is stated in Lemma C.3. To prove the induction step we make a case distinction to eliminate the $\exists$-quantification over $\mathfrak{h}_n$ and then we perform another inner induction proof over pairs $\langle l, t\rangle$.

C.1. Base Step: Initial Knowledge

Lemma C.2 considers soundness of knowledge in the initial state ($n = 0$). Since $t \leq n$ it holds that $t = 0$.

Lemma C.2 (Soundness of the initial state) *Let $\mathcal{D}$ be a domain description and $\delta_0 = \langle u_0, \Sigma_0\rangle$ a grounded valid initial c-state of $\mathcal{D}$ and h_0 be the initial h-state of D. Then (C.2) holds.*

$$\forall l : \mathfrak{h}_0 \models \langle l, 0\rangle \Rightarrow \Sigma_0^0 \models l \tag{C.2}$$

Proof:
Definition 3.2 concerning the initial h-state h_0, Definition 3.6 concerning the re-evaluated initial k-state and the definition of initial knowledge in (Son and Baral, 2001, p. 28, Definition 3) directly prove the Lemma. ∎

C.2. Induction Step: Knowledge Gain for Single State Transitions

The induction step reflects that after the $n+1$-th state transition is performed then there exists at least one h-state $\mathfrak{h}_{n+1}$ resulting from the state transition such that $\mathfrak{h}_{n+1} \models \langle l, t \rangle \Rightarrow \Sigma_{n+1}^{t} \models l$. This is formalized in Lemma C.3.

Lemma C.3 (Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$ for single state transitions) *Let $\mathfrak{h}_n \in \widehat{\Psi}(\alpha_n, \mathfrak{h}_0)$ be an h-state such that (C.3) holds. Then (C.4) holds as well.*

$$\forall l, t : (\mathfrak{h}_n \models \langle l, t \rangle) \Rightarrow (\Sigma_n^t \models l) \tag{C.3}$$

$$\begin{aligned} &\exists \mathfrak{h}_{n+1} \in \Psi(a_{n+1}, \mathfrak{h}_n) : \\ &\forall l, t : (\mathfrak{h}_{n+1} \models \langle l, t \rangle) \Rightarrow (\Sigma_{n+1}^t \models l) \end{aligned} \tag{C.4}$$

with $t \leq n+1$.

Proof:
We fist make some substitutions and generalize over possible sensing results to eliminate the $\exists$-quantification over h-states. This allows us to perform induction over pairs $\langle l, t \rangle$.

(C.4)

Recall the $\mathcal{HPX}$-transition function (3.7).

$$\Psi(a_{n+1}, \mathfrak{h}_n) = \bigcup_{k \in sense(\{a_{n+1}\}, \mathfrak{h}_n)} eval(\langle \boldsymbol{\alpha}_{n+1}, \boldsymbol{\kappa}_n \cup k \rangle)$$

where $\boldsymbol{\alpha}_{n+1} = \boldsymbol{\alpha}(\mathfrak{h}_n) \cup \{\langle a_{n+1}, now(\mathfrak{h}_n) \rangle\}$ and $\boldsymbol{\kappa}_n = \boldsymbol{\kappa}(\mathfrak{h}_n)$. Lemma C.4 states that $now(\mathfrak{h}_n) = n$. With this we substitute in (C.4) and obtain (C.5)

$$\begin{aligned} &\exists \mathfrak{h}_{n+1} \in \bigcup_{k \in sense(\{a_{n+1}\}, \mathfrak{h}_n)} eval(\langle \boldsymbol{\alpha}_{n+1}, \boldsymbol{\kappa}_n \cup k \rangle) : \\ &\forall l, t : (\mathfrak{h}_{n+1} \models \langle l, t \rangle) \Rightarrow (\Sigma_{n+1}^t \models l) \end{aligned} \tag{C.5}$$

with $t \leq n+1$.

(C.5)

To prove (C.5) we make a generalization which eliminates the ∃-quantification over $\mathfrak{h}_{n+1}$. To this end consider the $sense$ function (3.8) which we rewrite as (C.6).

$$sense(\{a_{n+1}\}, \mathfrak{h}_n) = \begin{cases} \{\{\langle f^s, t^s\rangle\}, \{\langle \neg f^s, t^s\rangle\}\} & \text{if } \mathcal{KP}^{a_{n+1}} = f^s \wedge \\ & \quad \{\langle f^s, t^s\rangle, \langle \neg f^s, t^s\rangle\} \cap \boldsymbol{\kappa}_n = \emptyset \\ \{\emptyset\} & \text{otherwise} \end{cases} \tag{C.6}$$

where $t^s = now(\mathfrak{h}_n)$. By Lemma C.4 it holds that $t^s = n$.
Let u_n denote the state which results from the application of the first n actions on the initial state u_0. This is formally expressed by (C.7).

$$u_n = Res(a_n, Res(a_{n-1}, \cdots Res(a_n, u_0))) \tag{C.7}$$

Since u_n is a set of fluent symbols there must be one sensing result $k \in sense(\{a_{n+1}\}, \mathfrak{h}_n)$ which corresponds to u_n. This correspondence is denoted by an auxiliary boolean function $corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n)$ (C.8).

$$\begin{aligned} & corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n) \Leftrightarrow \\ & \Big(k \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \wedge \\ & \big((k = \{\emptyset\}) \vee (k = \{\langle f^s, n\rangle\} \wedge f^s \in u_n) \vee (k = \{\langle \neg f^s, n\rangle\} \wedge f^s \notin u_n)\big)\Big) \end{aligned} \tag{C.8}$$

where $f^s = \mathcal{KP}^{a_{n+1}}$. Consequently, in order to show that (C.5) holds it is sufficient to show that (C.9) holds.

$$\begin{aligned} \forall l, t, k : & \big((eval(\langle \boldsymbol{\alpha}_{n+1}, \boldsymbol{\kappa}_n \cup k\rangle) \models \langle l, t\rangle) \\ & \Rightarrow (\Sigma_{n+1}^t \models l)\big) \end{aligned} \tag{C.9}$$

with $t \leq n+1$ and $corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n)$ holds.

To prove (C.9) we perform induction according to the structure of (B.16) in Lemma B.9 over pairs $\langle l, t\rangle$.

Let $\mathfrak{h}^k_{n+1} = eval(\langle \boldsymbol{\alpha}_{n+1}, \boldsymbol{\kappa}_n \cup k\rangle$ such that $corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n$ holds. Then (B.16) rewrites as follows:

$$\begin{aligned}\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \Leftrightarrow \Big(& \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n) \\ & \vee \{\langle l,t\rangle\} \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \\ & \vee \langle l,t\rangle \in add_{fwd}(\mathfrak{h}^k_{n+1}) \\ & \vee \langle l,t\rangle \in add_{back}(\mathfrak{h}^k_{n+1}) \\ & \vee \langle l,t\rangle \in add_{cause}(\mathfrak{h}^k_{n+1}) \\ & \vee \langle l,t\rangle \in add_{pd^{pos}}(\mathfrak{h}^k_{n+1}) \\ & \vee \langle l,t\rangle \in add_{pd^{neg}}(\mathfrak{h}^k_{n+1})\Big)\end{aligned}$$

From (B.16) we extract the set of implications (C.10).

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n) \quad \text{(C.10a)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \{\langle l,t\rangle\} \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \quad \text{(C.10b)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in add_{fwd}(\mathfrak{h}^k_{n+1}) \quad \text{(C.10c)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in add_{back}(\mathfrak{h}^k_{n+1}) \quad \text{(C.10d)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in add_{cause}(\mathfrak{h}^k_{n+1}) \quad \text{(C.10e)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in add_{pd^{pos}}(\mathfrak{h}^k_{n+1}) \quad \text{(C.10f)}$$

$$\forall \langle l,t\rangle : \langle l,t\rangle \in \boldsymbol{\kappa}^k_{n+1} \Leftarrow \langle l,t\rangle \in add_{pd^{neg}}(\mathfrak{h}^k_{n+1}) \quad \text{(C.10g)}$$

where $\boldsymbol{\kappa}^k_{n+1} = \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1})$.
Two implications generate knowledge about a pair $\langle l,t\rangle$ independently from other pairs $\langle l',t'\rangle$ with $\langle l,t\rangle \neq \langle l',t'\rangle$. These are the cases (C.10a) and (C.10b). Since a soundness proof for these cases does not rely on the induction hypothesis we consider these cases for the base steps.
The remaining implications (C.10c), (C.10d), (C.10e), (C.10f) and (C.10g) produce $\langle l,t\rangle$ but rely on knowledge about $\langle l',t'\rangle$ with $\langle l,t\rangle \neq \langle l',t'\rangle$. For example a pair $\langle l,t\rangle$ is produced by the forward inertia function $add_{fwd}(\mathfrak{h}^k_{n+1})$ if $\langle l,t-1\rangle$ is known to hold in $\mathfrak{h}^k_{n+1}$. These implications are considered in the induction step because proving soundness relies on the induction hypothesis. The induction is complete because (B.16) is a bi-implication, i.e. all pairs $\langle l,t\rangle$ are reached.

Base Step 1 – (C.10a)

Recall (C.9):

$$\forall l, t, k : (\mathfrak{h}_{n+1}^k \models \langle l, t \rangle) \Rightarrow (\Sigma_{n+1}^t \models l)$$

with $t \leq n+1$.

Consider (C.10a):

$\forall \langle l, t \rangle : \big(\mathfrak{h}_{n+1}^k \models \langle l, t \rangle \Leftarrow \langle l, t \rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n)\big)$

$$\forall l, t : \big((\langle l, t \rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n)) \Rightarrow (\Sigma_{n+1}^t \models l)\big) \tag{C.11}$$

with $t \leq n+1$.

By (C.3) the following holds:

$$\forall l, t : \big((\langle l, t \rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n)) \Rightarrow (\Sigma_n^t \models l)\big)$$

To show that (C.11) holds it is sufficient to show that (C.12) holds.

$$\forall l, t : \big((\Sigma_n^t \models l) \Rightarrow (\Sigma_{n+1}^t \models l)\big) \tag{C.12}$$

with $t \leq n+1$.

Lemma C.5 states that (C.12) is true and we have proven base step 1.

Base Step 2 – (C.10b)

Recall (C.9):

$$\forall l, t, k : (corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n) \wedge \mathfrak{h}^k_{n+1} \models \langle l, t \rangle) \Rightarrow (\Sigma^t_{n+1} \models l)$$

with $t \leq n + 1$.

Consider (C.10b):
$\forall \langle l, t \rangle : \mathfrak{h}^k_{n+1} \models \langle l, t \rangle \Leftarrow \langle l, t \rangle \in \boldsymbol{\kappa}(\mathfrak{h}_n)$

$$\forall l, t, k : \big(corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n) \wedge \{\langle l, t \rangle\} \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \Rightarrow (\Sigma^t_{n+1} \models l)\big) \tag{C.13}$$

with $t \leq n + 1$.

Reconsider $corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n)$ (C.8):

$$\begin{aligned}
&corrSense(k, a_{n+1}, \mathfrak{h}_n, u_n) \Leftrightarrow \\
&\Big(k \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \wedge \\
&\big((k = \{\emptyset\}) \vee (k = \{\langle f^s, n \rangle\} \wedge f^s \in u_n) \vee (k = \{\langle \neg f^s, n \rangle\} \wedge f^s \notin u_n)\big)\Big)
\end{aligned}$$

Reconsider the $sense$ function (C.6):

$$sense(\{a_{n+1}\}, \mathfrak{h}_n) = \begin{cases} \{\{\langle f^s, n \rangle\}, \{\langle \neg f^s, n \rangle\}\} & \text{if } \mathcal{KP}^{a_{n+1}} = f^s \wedge \\ & \quad \{\langle f^s, n \rangle, \langle \neg f^s, n \rangle\} \cap \boldsymbol{\kappa}_n = \emptyset \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

Consequently, if $\{\langle l, t \rangle\} \in sense(\{a_{n+1}\}, \mathfrak{h}_n)$ then one of the following cases is true:

1. $\langle l, t \rangle = \langle f^s, n \rangle \wedge f^s \in u_n)$
2. $\langle l, t \rangle = \langle \neg f^s, n \rangle \wedge f^s \in u_n)$

   where $f^s = \mathcal{KP}^a$.
   We have to show that (C.13) holds in both cases. For brevity we show only case 1 (C.14). Case 2 is analogous.

$$\forall l, t : \big(\langle l, t \rangle = \langle f^s, n \rangle \wedge f^s \in u_n) \wedge \{\langle l, t \rangle\} \in sense(\{a_{n+1}\}, \mathfrak{h}_n) \Rightarrow (\Sigma^t_{n+1} \models l)\big) \tag{C.14}$$

with $t \leq n + 1$ and $f^s = \mathcal{KP}^a$.

(C.14)

To show that (C.14) holds it is sufficient to show that (C.15) holds.

$$(f^s \in u_n) \Rightarrow (\Sigma_{n+1}^n \models f^s) \tag{C.15}$$

with $t \leq n+1$ and $f^s = \mathcal{KP}^a$.

By transition function for sensing actions (3.26):

$$(f^s \in u_n) \Rightarrow \big(\Sigma_{n+1} = \{s|(s \in \Sigma_n) \wedge (f^s \in s)\}\big)$$

It follows that (C.16) holds.

$$(f^s \in u_n) \Rightarrow (\forall s \in \Sigma_{n+1} : f^s \in s) \tag{C.16}$$

$$(\forall s \in \Sigma_{n+1} : f^s \in s) \Rightarrow (\Sigma_{n+1}^n \models f^s) \tag{C.17}$$

with $t \leq n+1$ and $f^s = \mathcal{KP}^a$.

By Lemma C.6: $\Sigma_{n+1} = \Sigma_{n+1}^{n+1}$

$$(\forall s \in \Sigma_{n+1}^{n+1} : f^s \in s) \Rightarrow (\Sigma_{n+1}^n \models f^s) \tag{C.18}$$

with $t \leq n+1$ and $f^s = \mathcal{KP}^a$.

By (3.29): $\Sigma_{n+1}^{n+1} = \bigcup\limits_{s \in \Sigma_{n+1}^n}^{R} es(a_{n+1}, s)$

Recall that we restrict that sensing actions do not have effect propositions. Since $\mathcal{KP}^{a_{n+1}} = f^s$ it holds that a_{n+1} is a sensing action and has no effect propositions. Therefore, by the $\mathcal{A}_k$ result function (3.25):

$$\forall s \in \Sigma_{n+1}^n : Res(a_{n+1}, s) = s$$

And consequently $\Sigma_{n+1}^n = \Sigma_{n+1}^{n+1}$.

$$(\forall s \in \Sigma_{n+1}^n : f^s \in s) \Rightarrow (\Sigma_{n+1}^n \models f^s) \tag{C.19}$$

with $t \leq n+1$ and $f^s = \mathcal{KP}^a$.

By definition of $\models$ (3.6): $\Sigma_{n+1}^n \models f^s \Leftrightarrow (\forall s \in \Sigma_{n+1}^n : f^s \in s)$.

This shows that (C.19) is true. Therefore base step 2 (C.11) is true.

Induction Step 1 – (C.10c)

Recall (C.9):

$$\forall l, t, k : (\mathfrak{h}^k_{n+1} \models \langle l, t\rangle) \Rightarrow (\Sigma^t_{n+1} \models l)$$

with $t \leq n+1$.

Consider (C.10c):

$\forall \langle l, t\rangle : \mathfrak{h}^k_{n+1} \models \langle l, t\rangle \Leftarrow \langle l, t\rangle \in add_{fwd}(\mathfrak{h}^k_{n+1})$

$$\forall l, t, k : \big(\langle l, t\rangle \in add_{fwd}(\mathfrak{h}^k_{n+1}) \Rightarrow (\Sigma^t_{n+1} \models l)\big) \tag{C.20}$$

with $t \leq n+1$.

Consider the definition of add_{fwd} (3.11):

$$\begin{aligned} add_{fwd}(\mathfrak{h}^k_{n+1}) = \{\langle l, t\rangle \,|(\langle l, t-1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \wedge \\ inertial(l, t-1, \mathfrak{h}^k_{n+1}) \wedge t \leq now(\mathfrak{h}^k_{n+1}))\} \end{aligned}$$

By Lemma C.4 it holds that $now(\mathfrak{h}^k_{n+1}) = n+1$. To prove that (C.20) holds we prove that (C.21) holds.

$$\forall l, t, k : \Big(\big(\langle l, t-1\rangle \in \boldsymbol{\kappa}^k_{n+1} \wedge inertial(l, t-1, \mathfrak{h}^k_{n+1})\big) \Rightarrow (\Sigma^t_{n+1} \models l)\Big) \tag{C.21}$$

with $t \leq n+1$.

Consider the definition of $inertial$ (3.10):

$$\begin{aligned} inertial(l, t-1, \mathfrak{h}^k_{n+1}) \Leftrightarrow \forall \langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) : \\ \big(e(ep) = \bar{l}\big) \Rightarrow \Big(\exists l^c \in c(ep) : \big\langle \overline{l^c}, t-1\big\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1})\Big) \end{aligned}$$

To prove that (C.21) holds we prove that (C.22) holds.

$$\begin{aligned} \forall l, t, k : \Big(\big(\langle l, t-1\rangle \in \boldsymbol{\kappa}^k_{n+1} \wedge \\ \big(\forall ep : \big(\langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) \Rightarrow \\ \big((e(ep) \neq \bar{l}) \vee (\exists l^c \in c(ep) : \big\langle \overline{l^c}, t-1\big\rangle \in \boldsymbol{\kappa}^k_{n+1})\big)\big)\big) \\ \Rightarrow (\Sigma^t_{n+1} \models l)\Big) \end{aligned} \tag{C.22}$$

with $t \leq n+1$.

(C.22)

By induction hypothesis: $(\langle l, t-1\rangle \in \boldsymbol{\kappa}_{n+1}^k) \Rightarrow (\Sigma_{n+1}^{t-1} \models l)$.

By definition of $\models$ (3.23): $\left(\Sigma_{n+1}^{t-1} \models l\right) \Rightarrow \left(\forall s \in \Sigma_{n+1}^{t-1} : s \models l\right)$.

$$\begin{aligned}\forall l,t,k:\Big(&\big((\forall s \in \Sigma_{n+1}^{t-1} : s \models l)\wedge \\ &\big(\forall ep : \big(\langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}_{n+1}^k) \Rightarrow \\ &\quad\big((e(ep) \neq \bar{l}) \vee (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1\rangle \in \boldsymbol{\kappa}_{n+1}^k)\big)\big)\big) \\ &\Rightarrow (\Sigma_{n+1}^t \models l)\Big)\end{aligned} \tag{C.23}$$

with $t \leq n+1$.

By the definition of effect histories (3.3) and the extended transition function (3.18):

$$\langle ep, t-1\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}_{n+1}^k) \Rightarrow (ep \in \mathcal{EP}^{a_t})$$

To show that (C.23) holds it is sufficient to show that (C.24) holds.

$$\begin{aligned}\forall l,t,k:\Big(&\big(\forall s \in \Sigma_{n+1}^{t-1} : s \models l\big)\wedge \\ &\big(\forall ep \in \mathcal{EP}^{a_t} : \big((e(ep) \neq \bar{l}) \vee (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1\rangle \in \boldsymbol{\kappa}_{n+1}^k)\big)\big)\big) \\ &\Rightarrow (\Sigma_{n+1}^t \models l)\Big)\end{aligned} \tag{C.24}$$

with $t \leq n+1$.

By definition of re-evaluated k-states (3.29): $\Sigma_{n+1}^t = \bigcup_{s \in \Sigma_{n+1}^{t-1}} Res(a_t, s)$.

By definition of $\models$ (3.23): $\left(\bigcup_{s \in \Sigma_{n+1}^{t-1}} Res(a_t, s) \models l\right) \Rightarrow (\forall s \in \Sigma_{n+1}^{t-1} : Res(a_t, s) \models l)$.

$$\begin{aligned}\forall l,t,k:\Big(&\big(\forall s \in \Sigma_{n+1}^{t-1} : s \models l\big)\wedge \\ &\big(\forall ep \in \mathcal{EP}^{a_t} : \big((e(ep) \neq \bar{l}) \vee (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1\rangle \in \boldsymbol{\kappa}_{n+1}^k)\big)\big)\big) \\ &\Rightarrow (\forall s \in \Sigma_{n+1}^{t-1} : Res(a_t, s) \models l)\Big)\end{aligned} \tag{C.25}$$

with $t \leq n+1$.

Case Distinction

To prove (C.25) we consider two cases for effect propositions ep, namely $e(ep) \neq \bar{l}$ and $\exists l^c \in c(ep) : \langle \overline{l^c}, t \rangle \in \boldsymbol{\kappa}^k_{n+1}$. We show that (C.25) holds in both cases.

1. $e(ep) \neq \bar{l}$ (Effect propositions do not have a complementary effect literal.)

(C.25)

We consider only cases where (C.26) holds.

$$e(ep) \neq \bar{l} \tag{C.26}$$

This simplifies (C.25) to (C.27).

$$\begin{aligned} \forall l, t : \Big(& \big(\forall s \in \Sigma^{t-1}_{n+1} : s \models l\big) \wedge \\ & \big(\forall ep \in \mathcal{EP}^{a_t} : e(ep) \neq \bar{l}\big) \\ & \Rightarrow (\forall s \in \Sigma^{t-1}_{n+1} : Res(a_t, s) \models l)\Big) \end{aligned} \tag{C.27}$$

with $t \leq n+1$.

Recall the $\mathcal{A}_k$ result function (3.25):

$$\begin{aligned} Res(a_t, s) &= s \cup E^+_{a_t}(s) \setminus E^-_{a_t}(s) \text{ where} \\ E^+_{a_t}(s) &= \{f | \exists ep \in \mathcal{EP}^{a_t} : e(ep) = f \wedge s \models c(ep)\} \\ E^-_{a_t}(s) &= \{f | \exists ep \in \mathcal{EP}^{a_t} : e(ep) = \neg f \wedge s \models c(ep)\} \end{aligned}$$

We distinguish two cases:

a) $l = f \wedge \forall ep \in \mathcal{EP}^{a_t} : (e(ep) \neq \neg f)$
In this case $E^-_{a_t}(s) = \emptyset$. Therefore: $\forall s \in \Sigma^{t-1}_{n+1} : (s \models f \Rightarrow Res(a_t, s) \models f)$

b) $l = \neg f \wedge \forall ep \in \mathcal{EP}^{a_t} : (e(ep) \neq f)$
In this case $E^+_{a_t}(s) = \emptyset$. Therefore: $\forall s \in \Sigma^{t-1}_{n+1} : (s \models \neg f \Rightarrow Res(a_t, s) \models \neg f)$

Consequently:

$$\begin{aligned} \forall l, t : \Big(& \big(\forall ep \in \mathcal{EP}^{a_t} : (e(ep) \neq \bar{l})\big) \\ & \Rightarrow \big(\forall s \in \Sigma^{t-1}_{n+1} : (s \models l \Rightarrow Res(a_t, s) \models l)\big)\Big) \end{aligned} \tag{C.28}$$

It follows from (C.28) that (C.27) holds.

2. $(\exists l^c \in c(ep) : \langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k)$ (Effect propositions have a condition literal which is known not to hold.)

(C.25)

We consider only cases where (C.29) holds.

$$\exists l^c \in c(ep) : \langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k \tag{C.29}$$

This simplifies (C.25) to (C.30).

$$\begin{aligned} \forall l, t, k :\Big(& (\forall s \in \Sigma_{n+1}^{t-1} : s \models l) \wedge \\ & (\forall ep \in \mathcal{EP}^{a_t} : (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k)) \\ & \Rightarrow (\forall s \in \Sigma_{n+1}^{t-1} : Res(a_t, s) \models l)\Big) \end{aligned} \tag{C.30}$$

with $t \leq n+1$.

Recall the $\mathcal{A}_k$ result function (3.25):

$Res(a_t, s) = s \cup E_{a_t}^+(s) \setminus E_{a_t}^-(s)$ where

$E_{a_t}^+(s) = \{f | \exists ep \in \mathcal{EP}^{a_t} : e(ep) = f \wedge s \models c(ep)\}$

$E_{a_t}^-(s) = \{f | \exists ep \in \mathcal{EP}^{a_t} : e(ep) = \neg f \wedge s \models c(ep)\}$

We distinguish two cases:

a) $l = f \wedge \forall ep \in \mathcal{EP}^{a_t} : (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k)$

By induction hypothesis we derive the following:

$$\forall l^c \in c(ep) : ((\langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k) \Rightarrow \Sigma_{n+1}^{t-1} \models \overline{l^c})$$

In this case clearly $\forall s \in \Sigma_{n+1}^{t-1} : E_{a_t}^-(s) = \emptyset$.
Therefore: $\forall s \in \Sigma_{n+1}^{t-1} : (s \models f \Rightarrow Res(a_t, s) \models f)$.

b) $l = \neg f \wedge \forall ep \in \mathcal{EP}^{a_t} : (\exists l^c \in c(ep) : \langle \overline{l^c}, t-1 \rangle \in \boldsymbol{\kappa}_{n+1}^k)$
It follows similarly to case a) that $E_{a_t}^+(s) = \emptyset$.
Therefore: $\forall s \in \Sigma_{n+1}^{t-1} : (s \models \neg f \Rightarrow Res(a_t, s) \models \neg f)$

From a) and b) follows:

$$\begin{aligned} \forall l, t :\Big(& (\forall ep \in \mathcal{EP}^{a_t} : (e(ep) \neq \bar{l})) \\ & \Rightarrow (\forall s \in \Sigma_{n+1}^{t-1} : (s \models l \Rightarrow Res(a_t, s) \models l))\Big) \end{aligned} \tag{C.31}$$

It follows from (C.31) that (C.30) holds.

Induction Step 2 – (C.10d)

This is analogous to induction step 1 – (C.10c).

Induction Step 3 – (C.10e)

This is analogous to induction step 4 – (C.10f).

Induction Step 4 – (C.10f)

Recall (C.9):

$$\forall l, t, k : (\mathfrak{h}^k_{n+1} \models \langle l, t\rangle) \Rightarrow (\Sigma^t_{n+1} \models l)$$

with $t \leq n+1$.

Recall (C.10f):
$\forall \langle l, t\rangle : \mathfrak{h}^k_{n+1} \models \langle l, t\rangle \Leftarrow \langle l, t\rangle \in add_{pd^{pos}}(\mathfrak{h}^k_{n+1})$

$$\forall l, t, k : \big(\langle l, t\rangle \in add_{pd^{pos}}(\mathfrak{h}^k_{n+1}) \Rightarrow (\Sigma^t_{n+1} \models l)\big) \tag{C.32}$$

with $t \leq n+1$.

Consider the definition of $add_{pd^{pos}}$ (3.14):

$$\begin{aligned} add_{pd^{pos}}(\mathfrak{h}^k_{n+1}) =& \{\langle l^c, t\rangle \,|\, \exists \langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) : \\ & l^c \in c(ep) \wedge \langle l^e, t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \wedge \left\langle \overline{l^e}, t\right\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \\ & \wedge \Big(\forall \left\langle ep', t\right\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) : \big(ep' = ep \vee e(ep') \neq l^e\big)\Big) \end{aligned}$$

To prove that (C.32) holds we prove that (C.33) holds.

$$\begin{aligned} \forall l, t, k : \Big(\big(\exists ep : \big(\langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) \wedge e(ep) = l^e \wedge \\ & l \in c(ep) \wedge \langle l^e, t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \wedge \left\langle \overline{l^e}, t\right\rangle \in \boldsymbol{\kappa}(\mathfrak{h}^k_{n+1}) \\ & \wedge (\forall \left\langle ep', t\right\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}^k_{n+1}) : (ep' = ep \vee e(ep') \neq l^e)))\big) \\ \Rightarrow (\Sigma^t_{n+1} \models l)\Big) \end{aligned} \tag{C.33}$$

with $t \leq n+1$.

(C.33)

By the definition of effect histories (3.3) and the extended transition function (3.18) it holds that:

$$\langle ep, t\rangle \in \boldsymbol{\epsilon}(\mathfrak{h}_{n+1}^k) \Rightarrow (ep \in \mathcal{EP}^{a_{t+1}})$$

To prove that (C.33) holds we prove that (C.34) holds.

$$\begin{aligned}\forall l, t, k : \Big(\big(\exists ep : \big(& ep \in \mathcal{EP}^{a_{t+1}} \wedge \wedge e(ep) = l^e \wedge \\ & l \in c(ep) \wedge \langle l^e, t+1\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{n+1}^k) \wedge \langle \overline{l^e}, t\rangle \in \boldsymbol{\kappa}(\mathfrak{h}_{n+1}^k) \\ & \wedge (\forall ep' \in \mathcal{EP}^{a_{t+1}} : (ep' = ep \vee e(ep') \neq l^e)))\big) \\ \Rightarrow (\Sigma_{n+1}^t \models l)\Big)\end{aligned} \tag{C.34}$$

with $t \leq n+1$.

By induction hypothesis:

$$(\langle l^e, t\rangle \in \boldsymbol{\kappa}_{n+1}^k) \Rightarrow (\Sigma_{n+1}^t \models l^e)$$

By definition of $\models$ (3.23):

$$\big(\Sigma_{n+1}^t \models l^e\big) \Rightarrow \big(\forall s \in \Sigma_{n+1}^t : s \models l^e\big)$$

$$\begin{aligned}\forall l, t : \Big(\big(\exists ep : \big(& ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge \\ & l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^{t+1} : s \models l^e) \wedge (\forall s \in \Sigma_{n+1}^t : s \models \overline{l^e}) \\ & \wedge (\forall ep' \in \mathcal{EP}^{a_{t+1}} : (ep' = ep \vee e(ep') \neq l^e)))\big) \\ \Rightarrow (\Sigma_{n+1}^t \models l)\Big)\end{aligned} \tag{C.35}$$

with $t \leq n+1$.

(C.35)

Recall the definition of re-evaluated k-states (3.29):

$$\Sigma_{n+1}^{t+1} = \bigcup_{s \in \Sigma_{n+1}^{t}} Res(a_{t+1}, s)$$

To show that (C.35) holds it is sufficient to show that (C.36) holds.

$$\forall l, t : \Big(\big(\exists ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^{t} : Res(a_{t+1}, s) \models \overline{l^e}) \wedge (\forall s \in \Sigma_{n+1}^{t} : s \models l^e) \wedge (\forall ep' \in \mathcal{EP}^{a_{t+1}} : (ep' = ep \vee e(ep') \neq l^e)) \big) \big) \Rightarrow (\Sigma_{n+1}^{t} \models l) \Big) \tag{C.36}$$

with $t \leq n+1$.

Simplification.

$$\forall l, t : \Big(\big(\exists ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^{t} : (Res(a_{t+1}, s) \models l^e \wedge s \models \overline{l^e})) \wedge (\forall ep' \in \mathcal{EP}^{a_{t+1}} : (ep' = ep \vee e(ep') \neq l^e)) \big) \big) \Rightarrow (\Sigma_{n+1}^{t} \models l) \Big) \tag{C.37}$$

with $t \leq n+1$.

We simplify:

$$\big(\exists ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge (\forall ep' \in \mathcal{EP}^{a_{t+1}} : (ep' = ep \vee e(ep') \neq l^e)) \big) \big)$$
$$\Leftrightarrow$$
$$\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \big) \big)$$

$$\forall l, t : \Big(\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^{t} : (Res(a_{t+1}, s) \models l^e \wedge s \models \overline{l^e})) \big) \big) \Rightarrow (\Sigma_{n+1}^{t} \models l) \Big) \tag{C.38}$$

with $t \leq n+1$.

(C.38)

Recall the $\models$ operator for k-states (3.23):

$$(\Sigma_{n+1}^t \models l) \Rightarrow (\forall s \in \Sigma_{n+1}^t : s \models l)$$

$$\forall l, t : \Big(\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = l^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : (Res(a_{t+1}, s) \models l^e \wedge s \models \overline{l^e}))\big)\big) \Rightarrow (\forall s \in \Sigma_{n+1}^t : s \models l) \Big) \tag{C.39}$$

with $t \leq n+1$.

For brevity we consider only the case where $e(ep) = l^e = f^e$. The case for $l^e = \neg f^e$ is similar. Recall (3.22):

$$s \models f^e \Leftrightarrow f^e \in s$$

$$\forall l, t : \Big(\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : (f^e \in Res(a_{t+1}, s) \wedge f^e \notin s))\big)\big) \Rightarrow (\forall s \in \Sigma_{n+1}^t : s \models l) \Big) \tag{C.40}$$

with $t \leq n+1$.

(C.40)

Recall the $\mathcal{A}_k$ result function (3.25):

$$\begin{aligned}Res(a_{t+1}, s) &= s \cup E^+_{a_{t+1}}(s) \setminus E^-_{a_{t+1}}(s) \text{ where}\\ E^+_{a_{t+1}}(s) &= \{f | \exists ep \in \mathcal{EP}^{a_{t+1}} : e(ep) = f \wedge s \models c(ep)\}\\ E^-_{a_{t+1}}(s) &= \{f | \exists ep \in \mathcal{EP}^{a_{t+1}} : e(ep) = \neg f \wedge s \models c(ep)\}\end{aligned}$$

Since we only consider cases with a positive effect literal $e(ep) = f^e$ the lower term $E^-_{a_{t+1}}(s)$ can be neglected. Consequently:

$$Res(a_{t+1}, s) = s \cup \{f | \exists ep \in \mathcal{EP}^{a_{t+1}} : e(ep) = f \wedge s \models c(ep)\} \tag{C.41}$$

We substitute (C.41) in (C.40) and obtain (C.42).

$$\begin{aligned}\forall l, t : \Big(\big(\exists! ep : \big(& ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge \\ & \big(\forall s \in \Sigma^t_{n+1} : \\ & \quad (f^e \in (s \cup \{f | \exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f \wedge s \models c(ep''))\}) \wedge f^e \not\in s) \big) \big) \big) \\ & \Rightarrow (\forall s \in \Sigma^t_{n+1} : s \models l) \Big)\end{aligned} \tag{C.42}$$

with $t \leq n + 1$.

We simplify:

$$\begin{aligned}&(f^e \in (s \cup \{f | \exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f \wedge s \models c(ep''))\}) \wedge f^e \not\in s)\\ &\Leftrightarrow (\exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f^e \wedge s \models c(ep'')))\end{aligned}$$

$$\begin{aligned}\forall l, t : \Big(\big(\exists ep! : \big(& ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge \\ & (\forall s \in \Sigma^t_{n+1} : (\exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f^e \wedge s \models c(ep''))))) \big) \\ & \Rightarrow (\forall s \in \Sigma^t_{n+1} : s \models l) \Big)\end{aligned} \tag{C.43}$$

with $t \leq n + 1$.

(C.43)

By definition of $\models$ (3.22):

$$(s \models c(ep'') \wedge l \in c(ep'')) \Rightarrow s \models l$$

$$\forall l, t : \Big(\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : (\exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f^e \wedge s \models l)))) \big) \Rightarrow (\forall s \in \Sigma_{n+1}^t : s \models l) \Big) \tag{C.44}$$

with $t \leq n + 1$.

We simplify.

$$\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : (\exists ep'' \in \mathcal{EP}^{a_{t+1}} : (e(ep'') = f^e \wedge s \models l)))) \big) \Leftrightarrow \big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : s \models l) \big) \big)$$

$$\forall l, t : \Big(\big(\exists! ep : \big(ep \in \mathcal{EP}^{a_{t+1}} \wedge e(ep) = f^e \wedge l \in c(ep) \wedge (\forall s \in \Sigma_{n+1}^t : s \models l) \big) \big) \Rightarrow (\forall s \in \Sigma_{n+1}^t : s \models l) \Big) \tag{C.45}$$

with $t \leq n + 1$.

It is easy to see that (C.45) is true.

Induction Step 5 – (C.10g)

This is analogous to induction step 4 – (C.10f).

■

C.3. Additional Lemmata

Number of Steps

The following Lemma C.4 concerns the number of state transitions for an h-state $\mathfrak{h}$.

Lemma C.4 (Step number for sequences of actions) *Given a domain $\mathcal{D}$ with an initial h-state $\mathfrak{h}_0$ and a sequence of actions $\alpha_n = [a_1||\cdots||a_n]$. Then the following holds:*

$$\forall \mathfrak{h} \in \widehat{\Psi}(\alpha_n, \mathfrak{h}_0) : now(\mathfrak{h}) = n \tag{C.46}$$

Proof:
The Lemma follows directly from the extended $\mathcal{HPX}$-transition function (3.18) and the $\mathcal{HPX}$-transition function (3.7).

Knowledge Persistence

The following Lemmata state that in the temporal query semantics $\mathcal{A}_k{}^{TQS}$, knowledge itself is persistent (Lemma C.5) an that knowledge about the presence is no affected by re-evaluation (Lemma C.6).

Lemma C.5 (Knowledge persistence in re-evaluated c-states) *Given a domain $\mathcal{D}$, a valid initial c-state $\delta_0 = \langle u_0, \Sigma_0 \rangle$ a sequence of actions $\alpha = [a_1, \ldots, a_n, a_{n+1}]$ which produces re-evaluated k-states Σ_n^t and Σ_{n+1}^t according to Definition 3.7. Then (C.47) holds.*

$$\Sigma_n^t \models l \Rightarrow \Sigma_{n+1}^t \models l \tag{C.47}$$

with $0 \leq t \leq n$.

Proof: For brevity we only consider positive literals, i.e. $l = f$. We make a case distinction concerning sensing and non-sensing actions:

1. If a_{n+1} is a non-sensing action then transition function (3.24) evaluates as follows: $\Phi(a_{n+1}, \langle u_n, \Sigma_n \rangle) = \langle u_{n+1}, \Sigma_{n+1} \rangle$ with $u_{n+1} = Res(a_{n+1}, u_n)$ and $\Sigma_{n+1} = \{Res(a_{n+1}, s_n) | s_n \in \Sigma_n\}$. With this and Definition 3.6 about re-evaluated initial k-states we conclude that $\Sigma_{n+1}^0 = \Sigma_n^0$. Hence, by Definition 3.7 about re-evaluated c-states it must be true that for an arbitrary f: If $\forall s \in \Sigma_n^t : f \in s$ then $\forall s \in \Sigma_{n+1}^t : f \in s$ and the Lemma is proven for the case of non-sensing actions.

2. If a_{n+1} is a sensing action then transition function (3.26) evaluates as $\Phi(a_{n+1}, \langle u_n, \Sigma_n \rangle) = \langle u_n, \{s_n | (s_n \in \Sigma_n) \wedge (f \in s_n \Leftrightarrow f \in u_n)\} \rangle$. For this reason $\Sigma_n \supseteq \Sigma_{n+1}$. Hence by Definition 3.6 it must be true that $\Sigma_n^0 \supseteq \Sigma_{n+1}^0$ and by Definition 3.7 it must be true that for an arbitrary f: If $\forall s \in \Sigma_n^t : f \in s$ then $\forall s \in \Sigma_{n+1}^t : f \in s$. The Lemma is proven for the case of sensing actions.

■

Lemma C.6 (Re-evaluation does not affect knowledge about the presence) *Given a domain $\mathcal{D}$, a valid initial c-state $\delta_0 = \langle u_0, \Sigma_0 \rangle$ and a sequence of actions $\alpha_n = [a_1; \ldots; a_n]$ such that $\langle u_n, \Sigma_n \rangle = \Phi(a_n, \Phi(a_{n-1}, \cdots \Phi(a_1, \langle u_0, \Sigma_0 \rangle)))$. Let $\Sigma_n^n = Res(a_n, Res(a_{n-1}, \cdots Res(a_1, \Sigma_n^0)))$ be a re-evaluated k-state with Σ_n^0 as the re-evaluated initial state according to Definition 3.6. Then (C.48) holds.*

$$\Sigma_n = \Sigma_n^n \tag{C.48}$$

Proof:
Induction over n. We show that given (C.48) holds (C.49) holds as well.

$$\Sigma_{n+1} = \Sigma_{n+1}^{n+1} \tag{C.49}$$

where $\langle u_{n+1}, \Sigma_{n+1} \rangle = \Phi(a_{n+1}, \Phi(a_n, \cdots \Phi(a_1, \langle u_0, \Sigma_0 \rangle)))$ is a c-state resulting from the application of the transition functions (3.24), (3.26) and $\Sigma_{n+1}^{n+1} = Res(a_{n+1}, Res(a_n, \cdots Res(a_1, \Sigma_{n+1}^0))$ is a re-evaluated k-state with Σ_{n+1}^0 as the re-evaluated initial state according to Definition 3.6.
Base Step: $\Sigma_0 = \Sigma_0^0$. This emerges from Definitions 3.2, 3.6 and 3.7. (Intuitively, if no action is applied then re-evaluation is not applicable.)
Induction Step: Given that (C.48) holds for one $n \geq 0$, then it holds that $\Sigma_{n+1} = \Sigma_{n+1}^{n+1}$. The re-evaluated c-state after $n+1$ actions is obtained with:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \Sigma_{n+1}^0} Res(a_{n+1}, Res(a_n, \cdots Res(a_1, s))) \tag{C.50}$$

We distinguish whether a_{n+1} is a sensing or non-sensing action:

1. a_{n+1} is a non-sensing action. In this case, according to the transition function (3.24) it holds that

   $$\forall s : (s \in \Sigma_n \Leftrightarrow Res(a_{n+1}, s) \in \Sigma_{n+1})$$

   By definition of re-evaluated initial k-states (3.29) it follows that (C.51) holds.

   $$\Sigma_n^0 = \Sigma_{n+1}^0 \tag{C.51}$$

   Substituting (C.51) in (C.50) yields:

   $$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \Sigma_n^0} Res(a_{n+1}, Res(a_n, \cdots Res(a_1, s))) \tag{C.52}$$

By definition of re-evaluated k-states (3.29) it holds that $\Sigma_n^n = \bigcup_{s \in \Sigma_n^0} Res(a_n, \cdots Res(a_1, s))$ and we can rewrite (C.52) as:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \Sigma_n^n} Res(a_{n+1}, s) \tag{C.53}$$

By induction hypothesis we can substitute Σ_n^n with Σ_n and have:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \Sigma_n} Res(a_{n+1}, s) \tag{C.54}$$

We reformulate (C.54) as follows:

$$\Sigma_{n+1}^{n+1} = \{Res(a_{n+1}, s) | s \in \Sigma_n\} \tag{C.55}$$

The transition function (3.24) for non-sensing actions is:

$$\begin{aligned} \Phi(a_{n+1}, \langle u_n, \Sigma_n \rangle) &= \langle u_{n+1}, \Sigma_{n+1} \rangle \\ &= \langle Res(a_{n+1}, u_n), \{Res(a_{n+1}, s) | s \in \Sigma_n\} \rangle \end{aligned}$$

It must therefore hold that

$$\Sigma_{n+1} = \{Res(a_{n+1}, s) | s \in \Sigma_n\} \tag{C.56}$$

It follows from (C.55) and (C.56) that $\Sigma_{n+1}^{n+1} = \Sigma_{n+1}$.

2. a_{n+1} is a sensing action with an arbitrary knowledge proposition $\mathcal{KP}^{a_{n+1}} = f^s$.

   We make another case distinction:

   a) $f^s \in u_n$:

      According to 3.29) the re-evaluated k-state Σ_{n+1}^0 is:

      $$\Sigma_{n+1}^0 = \{s \in \Sigma_0 | Res(a_{n+1}, Res(a_n, \cdots Res(a_1, s))) \in \Sigma_{n+1}\} \tag{C.57}$$

      If a_{n+1} is a sensing action, then $Res(a_{n+1}, s) = s$, and hence:

      $$\Sigma_{n+1}^0 = \{s \in \Sigma_0 | Res(a_n, \cdots Res(a_1, s)) \in \Sigma_{n+1}\} \tag{C.58}$$

      Given that a_{n+1} has a the knowledge proposition $\mathcal{KP}^{a_{n+1}} = f^s$, and $f^s \in u_n$, then from transition function (3.26) we can conclude that:

      $$\Sigma_{n+1} = \{s \in \Sigma_n | f^s \in s\} \tag{C.59}$$

Substituting (C.59) in (C.58) yields:

$$\Sigma_{n+1}^0 = \{s \in \Sigma_0 | Res(a_n, \cdots Res(a_1, s)) \in \{s' \in \Sigma_n | f^s \in s'\}\} \quad \text{(C.60)}$$

By Definition 3.6, the re-evaluated k-state Σ_n^0 is:

$$\Sigma_n^0 = \{s \in \Sigma_0 | Res(a_n, \cdots Res(a_1, s)) \in \Sigma_n\} \quad \text{(C.61)}$$

With (C.60) and (C.61) we can conclude that:

$$\Sigma_{n+1}^0 = \{s \in \Sigma_n^0 | f^s \in Res(a_n, \cdots Res(a_1, s))\} \quad \text{(C.62)}$$

Substituting (C.62) in (C.50) yields:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \{s_0 \in \Sigma_n^0 | f^s \in Res(a_n, \cdots Res(a_1, s_0))\}} Res(a_{n+1}, Res(a_n, \cdots Res(a_1, s))) \quad \text{(C.63)}$$

For a sensing actions a, it holds that $Res(a, s) = s$ for an arbitrary state s. Hence we write:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \{s_0 \in \Sigma_n^0 | f^s \in Res(a_n, \cdots Res(a_1, s_0))\}} Res(a_n, \cdots Res(a_1, s)) \quad \text{(C.64)}$$

We rewrite (C.64) and separate the union operator as follows:

$$\begin{aligned} \Sigma_{n+1}^{n+1} = & \bigcup_{s \in \Sigma_n^0} Res(a_n, \cdots Res(a_1, s)) \\ \setminus & \bigcup_{s \in \{s_0 \in \Sigma_n^0 | f^s \notin Res(a_n, \cdots Res(a_1, s_0))\}} Res(a_n, \cdots Res(a_1, s)) \end{aligned} \quad \text{(C.65)}$$

With the re-evaluation function (3.28) and the induction hypothesis (C.48) we have that

$$\begin{aligned} & \bigcup_{s \in \{s_0 \in \Sigma_n^0 | f^s \notin Res(a_n, \cdots Res(a_1, s_0))\}} Res(a_n, \cdots Res(a_1, s)) \\ = & \{Res(a_n, \cdots Res(a_1, s)) | s \in \Sigma_n^0 \wedge f^s \notin Res(a_n, \cdots Res(a_1, s))\} \\ \overset{(3.28)}{=} & \{s \in \Sigma_n^n | f^s \notin s\} \\ \overset{(C.48)}{=} & \{s \in \Sigma_n | f^s \notin s\} \end{aligned} \quad \text{(C.66)}$$

Thus, we can rewrite (C.65) as:

$$\Sigma_{n+1}^{n+1} = \bigcup_{s \in \Sigma_n^0} Res(a_n, \cdots Res(a_1, s)) \setminus \{s \in \Sigma_n | f^s \notin s\} \tag{C.67}$$

With the definition of re-evaluated c-states (3.28) it holds that $\Sigma_n^n = \bigcup_{s \in \Sigma_n^0} Res(a_n, \cdots Res(a_1, s))$ and we can rewrite (C.67) as:

$$\Sigma_{n+1}^{n+1} = \Sigma_n^n \setminus \{s \in \Sigma_n | f^s \notin s\} \tag{C.68}$$

By induction hypothesis we substitute Σ_n^n with Σ_n and obtain:

$$\Sigma_{n+1}^{n+1} = \Sigma_n \setminus \{s \in \Sigma_n | f^s \notin s\} \tag{C.69}$$

Given that $f^s \in u_n$, then the transition function 3.26 for sensing actions is:

$$\begin{aligned} \Phi(a_{n+1}, \langle u_n, \Sigma_n \rangle) &= \langle u_{n+1}, \Sigma_{n+1} \rangle \\ &= \langle u_n, \{s \in \Sigma_n | f^s \in s\} \rangle \end{aligned} \tag{C.70}$$

Extracting Σ_{n+1} from (C.70) yields:

$$\Sigma_{n+1} = \{s \in \Sigma_n | f^s \in s\} \tag{C.71}$$

We rewrite this as:

$$\Sigma_{n+1} = \Sigma_n \setminus \{s \in \Sigma_n | f^s \notin s\} \tag{C.72}$$

And substitute (C.72) in (C.69) to obtain:

$$\Sigma_{n+1}^{n+1} = \Sigma_{n+1} \tag{C.73}$$

b) $f^s \notin u_n$: Similar to the case where $f^s \in u_n$.

We have shown that for both sensing and non-sensing actions the induction step holds. This proves the Lemma. ■

Source Code and Examples

D.1. Foundational Theory of the Offline ASP Formalization of $\mathcal{HPX}$

The foundational part of the ASP formalization is provided in Listing D.1. Note that for brevity we use a predicate `l/1` instead of `literal/1` to denote literal declarations, and similarly `f/1` instead of `fluent/1`.

Listing D.1: Foundational theory (Γ_{hapx}) of the ASP implementation of $\mathcal{HPX}$

```
1  ▶ F1. Auxiliaries (Γaux)
2  s(0..maxS).
3  br(0..maxBr).
4  neq(B,B1) :- B != B1, br(B), br(B1).
5  l(neg(F)):- f(F).
6  l(F):- f(F).
7  complement(neg(F),F) :- f(F).
8  complement(L1,L2) :- complement(L2,L1).
9
10 ▶ F2. Concurrency (Γconc)
11 apply(EP,N,B) :- hasEP(A,EP), occ(A,N,B).
12 :- apply(EP1,T,B), hasEff(EP1,L), apply(EP2,T,B), hasEff(EP2,L),
       EP1 != EP2, br(B), l(L).
13 ▶ F3. Inertia (Γin)
14 kNotSet(L,T,N,B) :- not kMaySet(L,T,B), uBr(N,B), s(T),l(L).
15 kMaySet(L,T,B) :- apply(EP,T,B), hasEff(EP,L).
16 kNotSet(L,T,N,B) :- apply(EP,T,B), hasEff(EP,L), hasCond(EP,L1),
       knows(L2,T,N,B), complement(L1,L2),s(T).
17 knows(L,T,N,B) :- knows(L,T-1,N,B), kNotSet(L1,T-1,N,B),
       complement(L,L1), T<=N.
18 knows(L,T,N,B) :- knows(L,T+1,N,B), kNotSet(L,T,N,B), N > T.
19 knows(L,T,N,B) :- knows(L,T,N-1,B), uBr(N,B), N <= maxS.
```

```
20
21 ▶ F5.  Sensing and Branching (Γsense)
22 uBr(0,0).
23 sNextBr(N,B) :- sRes(L,N,B,B1).
24 uBr(N,B) :- uBr(N-1,B), not sNextBr(N-1,B), s(N).
25 kw(F,T,N,B) :- knows(F,T,N,B).
26 kw(F,T,N,B) :- knows(neg(F),T,N,B).
27 sRes(F,N,B,B):-occ(A,N,B),hasKP(A,F),not kw(neg(F),N,N,B),s(N).
28 1{sRes(neg(F),N,B,B1) : neq(B,B1)}1 :- occ(A,N,B), hasKP(A,F),
       not kw(F,N,N,B), s(N).
29 :- sRes(L,N,B,B1), uBr(N,B1), l(L), neq(B,B1).
30 :- 2{sRes(L,N,B,B1) :br(B) : l(L)},br(B1),s(N).
31 uBr(N,B1) :- sRes(L,N-1,B,B1), s(N).
32 knows(L,N-1,N,B1) :- sRes(L,N-1,B,B1), s(N).
33 :- 2{occ(A,N,B) : hasKP(A,_)}, br(B),s(N).
34 knows(L,T,N,B1) :- sRes(_,N-1,B,B1), knows(L,T,N-1,B), N>=T.
35 apply(EP,T,B1) :- sRes(_,N,B,B1), apply(EP,T,B), N>=T.
36
37 ▶ F4.  Inference Mechanisms (Γinfer)
38 knows(L,T,N,B) :- kCause(L,T,N,B), uBr(N,B).
39 knows(L,T,N,B) :- kPosPost(L,T,N,B), uBr(N,B).
40 knows(L,T,N,B) :- kNegPost(L,T,N,B), uBr(N,B).
41
42 ▶ F6.  Plan verification (Γverify)
43 notWG(N,B) :- wGoal(L), uBr(N,B), not knows(L,N,N,B), l(L).
44 allWGAchieved(N) :- not notWG(N,B), uBr(N,B).
45 :- not allWGAchieved(maxS).
46 notSG(N,B) :- sGoal(L), uBr(N,B), not knows(L,N,N,B), l(L).
47 :- notSG(maxS,B), uBr(maxS,B).
48 notGoal(N,B) :- notSG(N,B).
49 notGoal(N,B) :- notWG(N,B).
50
51 ▶ F7.  Plan generation and optimization (Γplan)
52 % Sequential Planning:
53 1{occ(A,N,B): a(A)}1 :- uBr(N,B), notGoal(N,B), N < maxS.
54 % Concurrent Planning:
55 % 1{occ(A,N,B) : a(A)} :- uBr(N,B), notGoal(N,B), N < maxS.
56 #minimize {occ(_,_,_) @ 1}.
```

D.2. Basic Postdiction Example: Driving Through a Door

Consider the domain description in Listing D.2.

```
(:action open_door :effect if ¬jammed then is_open)
(:action drive :effect when is_open in_liv)
(:action sense_open   :observe is_open)
(:init ¬is_open)
(:goal weak in_liv)
```

Listing D.2: Simplified problem of moving through a door

The following plan achieves the weak goal:

```
p1 = [open_door;
          sense_open;
         [if open
          then [drive]]
```

Listing D.3: Plan for problem in Listing D.2

According to Definition 3.2 about initial knowledge we have

$$\mathfrak{h}_0 = \langle \{\}, \{\langle \neg\texttt{is_open}, 0\rangle\}\rangle$$

We will now go investigate how the extended transition fuction (3.18) generates the transition tree if the plan p_1 is applied.

1. For the application of the first action `open_door` the extended transition function (3.18) rewrites as:

$$\widehat{\Psi}(p_1, \mathfrak{h}_0) = \bigcup_{\mathfrak{h}_1 \in \Psi(\texttt{open_door}, \mathfrak{h}_0)} \widehat{\Psi}(\texttt{sense_open;if is_open then [drive]}, \mathfrak{h}_1) \tag{D.1}$$

   Evaluating the application of action `open_door` with the transition function (3.7) yields that

$$\begin{aligned} &\Psi(\texttt{open_door}, \mathfrak{h}_0) = \{\mathfrak{h}_1\} \\ &\text{where } \mathfrak{h}_1 = \langle\{\langle \texttt{open_door}, 0\rangle\}, \{\langle\neg\texttt{is_open}, 0\rangle\}\rangle \end{aligned} \tag{D.2}$$

2. For the application of the second action `sense_open` we substitute (D.2) in (D.1) and the extended transition function (3.18) becomes

$$\begin{aligned} &\widehat{\Psi}(\texttt{[sense_open; [if is_open then [drive]]]}\mathfrak{h}_1) = \\ &\bigcup_{\mathfrak{h}_2 \in \Psi(\texttt{sense_open}, \mathfrak{h}_1)} \widehat{\Psi}(\texttt{if is_open then [drive]}, \mathfrak{h}_2) \end{aligned} \tag{D.3}$$

Evaluating `sense_open` with the transition function (3.7) yields that

$$\begin{aligned}
&\Psi(\texttt{sense_open}, \mathfrak{h}_1) = \{\mathfrak{h}_2^+, \mathfrak{h}_2^-\} \text{ where} \\
&\mathfrak{h}_2^+ = \Big\langle \{\langle \texttt{open_door}, 0\rangle, \langle \texttt{sense_open}, 1\rangle\}, \\
&\qquad \{\langle \neg\texttt{is_open}, 0\rangle, \langle \texttt{is_open}, 1\rangle, \langle \texttt{is_open}, 2\rangle, \\
&\qquad \langle \neg\texttt{jammed}, 0\rangle, \langle \neg\texttt{jammed}, 1\rangle, \langle \neg\texttt{jammed}, 2\rangle\}\Big\rangle \\
&\mathfrak{h}_2^- = \Big\langle \{\langle \texttt{open_door}, 0\rangle, \langle \texttt{sense_open}, 1\rangle\}, \\
&\qquad \{\langle \neg\texttt{is_open}, 0\rangle, \langle \neg\texttt{is_open}, 1\rangle, \langle \neg\texttt{is_open}, 2\rangle, \\
&\qquad \langle \texttt{jammed}, 0\rangle, \langle \texttt{jammed}, 1\rangle, \langle \texttt{jammed}, 2\rangle\}\Big\rangle
\end{aligned} \tag{D.4}$$

3. Now we consider two cases:

   a) $\mathfrak{h}_2^+$: The extended transition function (3.18) evaluates as

$$\begin{aligned}
&\widehat{\Psi}(\texttt{[if is_open then [drive]]}, \mathfrak{h}_2^+) = \\
&\quad \begin{cases} \widehat{\Psi}(\texttt{[drive]}\,\mathfrak{h}_2^+) & \text{if } \mathfrak{h}_2^+ \models \texttt{is_open} \\ \widehat{\Psi}(\texttt{[]}\,\mathfrak{h}_2^+) & \text{if } \mathfrak{h}_2^+ \not\models \texttt{is_open} \end{cases}
\end{aligned} \tag{D.5}$$

   With definition of the $\models$ operator (3.6) it is easy to see that $\mathfrak{h}_2^+ \models \texttt{is_open}$ is true, so we have that $\widehat{\Psi}(\texttt{[if is_open then [drive]]}\,\mathfrak{h}_2^+) = \widehat{\Psi}(\texttt{[drive]}\,\mathfrak{h}_2^+)$.

   b) $\mathfrak{h}_2^-$: The extended transition function (3.18) evaluates as

$$\begin{aligned}
&\widehat{\Psi}(\texttt{[if is_open then [drive]]}, \mathfrak{h}_2^+) = \\
&\quad \begin{cases} \widehat{\Psi}(\texttt{[drive]}\,\mathfrak{h}_2^-) & \text{if } \mathfrak{h}_2^+ \models \texttt{is_open} \\ \widehat{\Psi}(\texttt{[]}\,\mathfrak{h}_2^-) & \text{if } \mathfrak{h}_2^+ \not\models \texttt{is_open} \end{cases}
\end{aligned} \tag{D.6}$$

   With definition of the $\models$ (3.6) operator it is easy to see that $\mathfrak{h}_2^- \not\models \texttt{is_open}$ is true, so we have that $\widehat{\Psi}(\texttt{[if is_open then [drive]]}\,\mathfrak{h}_2^-) = \widehat{\Psi}(\texttt{[]}\,\mathfrak{h}_2^-)$.

4. a) $\mathfrak{h}_2^+$: With the extended transition function (3.18) it is easy to see that

$$\widehat{\Psi}(\texttt{[drive]}, \mathfrak{h}_2^+) = \{\Psi(\texttt{drive}, \mathfrak{h}_2^+)\} \tag{D.7}$$

   Evaluating the application of action `drive` with the transition function (3.7)

yields

$$\Psi(\texttt{drive}, \mathfrak{h}_2^+) = \{\mathfrak{h}_3^+\} \text{ where}$$
$$\mathfrak{h}_3^+ = \Big\langle \{\langle \texttt{open_door}, 0\rangle, \langle \texttt{sense_open}, 1\rangle, \langle \texttt{drive}, 2\rangle\}, \{\langle \neg\texttt{is_open}, 0\rangle, \langle \texttt{is_open}, 1\rangle, \langle \texttt{is_open}, 2\rangle, \langle \texttt{is_open}, 3\rangle, \langle \neg\texttt{jammed}, 0\rangle, \langle \neg\texttt{jammed}, 1\rangle, \langle \neg\texttt{jammed}, 2\rangle, \langle \neg\texttt{jammed}, 3\rangle, \langle \texttt{in_liv}, 3\rangle\}\Big\rangle \tag{D.8}$$

That is, $\mathfrak{h}_3^+ \models \langle \texttt{in_liv}, 3\rangle$, and according to the $solves$ function (3.20) the goal $l^{wg} = \texttt{in_liv}$ the goal is achieved.

b) $\mathfrak{h}_2^-$: For this case the extended transition function (3.18) directly produces

$$\widehat{\Psi}(\texttt{[]}, \mathfrak{h}_2^-) = \{\mathfrak{h}_2^-\} \tag{D.9}$$

This case does not affect the result of the planning problem, since according to (3.20) this is already solved due to $\mathfrak{h}_3^+ \models \langle \texttt{in_liv}, 3\rangle$.

D.3. Modifications For the Incremental Online ASP Implementation

The most important extension to the ASP formalization described in Chapter 4 is that we modify the Logic Program so that it is capable of *online planning*. To this end, we partition the Logic Program into $\#base$, $\#cumulative$ and $\#volatile$ parts and replace the variable N by the iterator t. We also have to replace the 3-ary $apply$ predicate by a 4-ary $apply$, because otherwise corresponding atoms would be produced multiple times during the grounding process. This is forbidden in incremental ASP (see (Gebser et al., 2011a) for details).

D.3.1. The Foundational Theory for Incremental Online Planning

The foundational theory for offline planning Γ_{hpx} is rewritten as an online planning theory Γ^o_{hpx}, presented in Listing D.4.In many cases modification are trivial: the variable N is replaced by the iterator t, and the 3-ary $apply(ep, t, b)$ is replaced by a 4-ary $apply(ep, T, t, b)$. The non-trivial modifications and extensions are described in Section 5.2.2.

Listing D.4: Domain independent part of the online implementation of $\mathcal{HPX}$ (Γ^o_{hpx})

```
1  ► FO1 Auxiliaries (Γ^o_aux)
2  #external exec/2.
3  #external sensed/2.
4  #external wGoal/1.
5  #external sGoal/1.
6  #cumulative t.
7  s(t).
8  #base.
9  br(0..maxB).
10 neq(B2,B2) :- B2 != B2, br(B2), br(B2).
11 l(neg(F)):- f(F).
12 l(F):- f(F).
13 complement(neg(F),F) :- f(F).
14 complement(L1,L2) :- complement(L2,L1).
15
16 ► FO2. Concurrency (Γ^o_conc)
17 #cumulative t.
18 apply(EP,t,t,B) :- hasEP(A,EP), occ(A,t,B).
19 :- apply(EP1,T,t,B), hasEff(EP1,L), apply(EP2,T,t,B), hasEff(EP2,L
       ), EP1 != EP2, br(B).
20 apply(EP,T,t,B) :- apply(EP,T,t-1,B).
21
22 ► FO3. Inertia (Γ^o_in)
23 #cumulative t.
24 kMaySet(L,T,t,B) :- apply(EP,T,t,B), hasEff(EP,L).
25 kNotSet(L,T,t,B) :- not kMaySet(L,T,t,B), uBr(t,B), s(T),l(L).
26 kNotSet(L,T,t,B) :- apply(EP,T,t,B), hasEff(EP,L), hasCond(EP,L1),
        knows(L2,T,t,B),complement(L1,L2),s(T), uBr(t,B).
27
28 knows(L,T,t,B) :- knows(L,T-1,t,B), kNotSet(L1,T-1,t,B),
       complement(L,L1), s(T),  br(B).
29 knows(L,T,t,B) :- knows(L,T+1,t,B), kNotSet(L,T,t,B) , s(T), br(B)
       , T < t.
30
31 knows(L,T,t,B) :- knows(L,T,t-1,B), uBr(t,B).
32
33 ► FO5. Sensing and Branching (Γ^o_sense)
34 #base.
35 uBr(0,0).
36 #cumulative t.
37 sNextBr(t-1,B1) :- sRes(L,t-1,B1,B2).
38 uBr(t,B) :- uBr(t-1,B), not sNextBr(t-1,B).
39
40 kw(F,T,t,B) :- knows(F,T,t,B).
41 kw(F,T,t,B) :- knows(neg(F),T,t,B).
42
43 sRes(F,t-1,B,B):- occ(A,t-1,B), hasKP(A,F), br(B), not sensed(neg(
       F), t-1), not kw(F, t-1, t-1, B).
```

```
44 1{sRes(neg(F),t-1,B1,B2) : neq(B1,B2)}1 :- occ(A,t-1,B1), hasKP(A,
       F), br(B1), not sensed(F,t-1), not kw(F, t-1, t-1, B1).
45
46 :- sRes(L,t-1,B1,B2), uBr(t-1,B2), l(L), neq(B1,B2).
47 :- 2{sRes(L,t-1,B1,B2) :br(B1) : l(L)},br(B2).
48
49 uBr(t,B2) :- sRes(L,t-1,B1,B2).
50 knows(L,t-1,t,B2) :- sRes(L,t-1,B1,B2).
51 :- 2{occ(A,t,B) : hasKP(A,_)}, br(B).
52
53 knows(L,T,t,B2) :- sRes(_,t-1,B1,B2), knows(L,T,t-1,B1), t >= T.
54 apply(EP,T,t,B2) :- sRes(_,t-1,B1,B2), apply(EP,T,t-1,B1), t >= T
55
```

56 ▶ **FO5. Plan verification** (**Γ^o_{verify}**)

```
57 #cumulative t.
58 notWG(t,B) :- wGoal(L), uBr(t,B), not knows(L,t,t,B), l(L).
59 allWGAchieved(t) :- not notWG(t,B), uBr(t,B).
60
61 #volatile t.
62 :- not allWGAchieved(t).
63
64 #cumulative t.
65 notSG(t,B) :- sGoal(L), uBr(t,B), not knows(L,t,t,B), l(L).
66
67 #volatile t.
68 :- notSG(t,B), uBr(t,B).
69
70 #cumulative t.
71 notGoal(t,B) :- notSG(t,B).
72 notGoal(t,B) :- notWG(t,B).
73
```

74 ▶ **FO7. Plan generation** (**Γ^o_{plan}**)

```
75 #cumulative t.
76 1{occ(A,t,B) :a(A)}1 :- uBr(t,B), not executedStep(t), notGoal(t,B
       ). % Sequential Planning
77 %1{occ(A,t,B) :a(A)} :- uBr(t,B), not executedStep(t), notGoal(t,B
       ). % Concurrent Planning
78
```

79 ▶ **FO8. Plan execution** (**Γ^o_{exec}**)

```
80 #cumulative t.
81 occ(A,t,B) :- exec(A,t), a(A), uBr(t,B).
82
83 executedStep(t) :- exec(A,t), a(A).
84 executedStep(t) :- sensed(L,t), l(L).
85
86 knows(L,t,t,B) :- sensed(L,t), uBr(t,B), l(L).
87 :- sensed(L1,t), uBr(t,B), knows(L2,t,t,B), complement(L1,L2).
88
```

89 ▶ **FO9. Abductive explanation** (**Γ^o_{abduct}**)

```
90  #cumulative t.
91  0{exoHappened(A,t-1,B) : hasEP(A,EP) : hasEff(EP,L1) : ea(A)}1 :-
        sensed(L1,t), uBr(t,B), knows(L2,t-1,t,B), complement(L1,L2).
92  apply(EP,t-1,t,B) :- hasEP(A,EP), exoHappened(A,t-1,B).
```

D.3.2. Incremental Modularity of $\mathcal{HPX}$-Logic Programs

The following Equations (D.10) – (D.18) incorporate both the domain-independent and the domain-specific theory and structures the generated incremental online $\mathcal{HPX}$-Logic Programs according to its `#base` (B), `#cumulative` ($P[t]$) and `#volative` ($Q[t]$) part, so that the resulting Logic Program is described by $R[t] = B \cup \bigcup_{0 \leq j \leq t} P[j] \cup Q[t]$ (2.42) (see Section 2.2.8).

The base part B is constituted by (D.10), and those LP rules generated by translation rules (TO1), (TO2), (TO3), (TO5), (TO7), (TO8). Rules (D.10) emerge mostly from auxiliary definitions Γ^o_{aux}.

$$br(0..\mathtt{maxB}) \tag{D.10a}$$

$$neq(B_2, B_2) \leftarrow B_2 \neq B_2, br(B_2), br(B_2) \tag{D.10b}$$

$$l(neg(F)) \leftarrow f(F) \tag{D.10c}$$

$$l(F) \leftarrow f(F) \tag{D.10d}$$

$$complement(neg(F), F) \leftarrow f(F) \tag{D.10e}$$

$$complement(L_1, L_2) \leftarrow complement(L_2, L_1) \tag{D.10f}$$

$$uBr(0, 0) \tag{D.10g}$$

The cumulative part $\bigcup_{0 \leq j \leq t} P[j]$ is described by equations (D.11) – (D.17), and those LP rules generated by translation rules (TO4) and (TO6). Equations (D.11) represent rules concerning concurrency (FO2) – (Γ^o_{conc}) (except for $s(t)$, which belongs to Γ^o_{aux}).

$$s(t) \tag{D.11a}$$

$$apply(EP, t, t, B) \leftarrow hasEP(A, EP), occ(A, t, B) \tag{D.11b}$$

$$\begin{aligned} \leftarrow & apply(EP_1, T, t, B), hasEff(EP_1, L), \\ & apply(EP_2, T, t, B), hasEff(EP_2, L), \\ & EP_1 \neq EP_2, br(B) \end{aligned} \tag{D.11c}$$

$$apply(EP, T, t, B) \leftarrow apply(EP, T, t-1, B) \tag{D.11d}$$

Equations (D.12) represent rules concerning inertia (FO3) – (Γ_in^o).

$$kMaySet(L,T,t,B) \leftarrow apply(EP,T,t,B), hasEff(EP,L) \tag{D.12a}$$
$$kNotSet(L,T,t,B) \leftarrow not\ kMaySet(L,T,t,B), uBr(t,B), s(T), l(L) \tag{D.12b}$$
$$kNotSet(L,T,t,B) \leftarrow apply(EP,T,t,B), hasEff(EP,L), hasCond(EP,L_1), knows(L_2,T,t,B), complement(L_1,L_2), s(T), uBr(t,B) \tag{D.12c}$$
$$knows(L,T,t,B) \leftarrow knows(L,T-1,t,B), kNotSet(L_1,T-1,t,B), complement(L,L_1), s(T), br(B) \tag{D.12d}$$
$$knows(L,T,t,B) \leftarrow knows(L,T+1,t,B), kNotSet(L,T,t,B), s(T), br(B), T < t \tag{D.12e}$$
$$knows(L,T,t,B) \leftarrow knows(L,T,t-1,B), uBr(t,B) \tag{D.12f}$$

Equations (D.14) represent rules concerning sensing and branching (FO5) – (Γ_sense^o).

$$sNextBr(t-1,B_1) \leftarrow sRes(L,t-1,B_1,B_2) \tag{D.13a}$$
$$uBr(t,B) \leftarrow uBr(t-1,B), not\ sNextBr(t-1,B) \tag{D.13b}$$
$$kw(F,T,t,B) \leftarrow knows(F,T,t,B) \tag{D.13c}$$
$$kw(F,T,t,B) \leftarrow knows(neg(F),T,t,B) \tag{D.13d}$$
$$sRes(F,t-1,B,B) \leftarrow occ(A,t-1,B), hasKP(A,F), br(B), not\ sensed(neg(F),t-1), not\ kw(F,t-1,t-1,B) \tag{D.13e}$$
$$1\{sRes(neg(F),t-1,B_1,B_2) : neq(B_1,B_2)\}1 \leftarrow occ(A,t-1,B_1), hasKP(A,F), br(B_1), not\ sensed(F,t-1), not\ kw(F,t-1,t-1,B_1) \tag{D.13f}$$
$$\leftarrow sRes(L,t-1,B_1,B_2), uBr(t-1,B_2), l(L), neq(B_1,B_2) \tag{D.13g}$$
$$\leftarrow 2\{sRes(L,t-1,B_1,B_2) : br(B_1) : l(L)\}, br(B_2) \tag{D.13h}$$
$$uBr(t,B_2) \leftarrow sRes(L,t-1,B_1,B_2) \tag{D.13i}$$
$$knows(L,t-1,t,B_2) \leftarrow sRes(L,t-1,B_1,B_2) \tag{D.13j}$$
$$\leftarrow 2occ(A,t,B) : hasKP(A,_), br(B) \tag{D.13k}$$
$$knows(L,T,t,B_2) \leftarrow sRes(_,t-1,B_1,B_2), knows(L,T,t-1,B_1), t \geq T \tag{D.13l}$$
$$apply(EP,T,t,B_2) \leftarrow sRes(_,t-1,B_1,B_2), apply(EP,T,t-1,B_1), t \geq T \tag{D.13m}$$

Equations (D.14) represent rules concerning plan verification (FO5) – (Γ_verify^o).

$$notWG(t,B) \leftarrow wGoal(L), uBr(t,B), not\ knows(L,t,t,B), l(L) \quad \text{(D.14a)}$$
$$allWGAchieved(t) \leftarrow not\ notWG(t,B), uBr(t,B) \quad \text{(D.14b)}$$
$$notSG(t,B) \leftarrow sGoal(L), uBr(t,B), not\ knows(L,t,t,B), l(L) \quad \text{(D.14c)}$$
$$notGoal(t,B) \leftarrow notSG(t,B) \quad \text{(D.14d)}$$
$$notGoal(t,B) \leftarrow notWG(t,B) \quad \text{(D.14e)}$$

Equations (D.15) represent rules concerning planning (FO7) – (Γ_plan^o).

$$1\{occ(A,t,B) : a(A)\}1 \leftarrow uBr(t,B), not\ executedStep(t), notGoal(t,B) \ \text{(Sequential Planning)} \quad \text{(D.15a)}$$
$$1\{occ(A,t,B) : a(A)\} \leftarrow uBr(t,B), not\ executedStep(t), notGoal(t,B) \text{(Concurrent Planning)} \quad \text{(D.15b)}$$

Equations (D.16) represent rules concerning action execution (FO8) – (Γ_exec^o).

$$occ(A,t,B) \leftarrow exec(A,t), a(A), uBr(t,B) \quad \text{(D.16a)}$$
$$executedStep(t) \leftarrow exec(A,t), a(A) \quad \text{(D.16b)}$$
$$executedStep(t) \leftarrow sensed(L,t), l(L) \quad \text{(D.16c)}$$
$$knows(L,t,t,B) \leftarrow sensed(L,t), uBr(t,B), l(L) \quad \text{(D.16d)}$$
$$\leftarrow sensed(L_1,t), uBr(t,B), knows(L_2,t,t,B), complement(L_1,L_2) \quad \text{(D.16e)}$$
$$\text{(D.16f)}$$

Equations (D.17) represent rules concerning exogenous events and abductive explanation (FO9) – (Γ_exo^o).

$$0\{exoHappened(A,t-1,B) : hasEP(A,EP) : hasEff(EP,L_1) : ea(A)\}1 \leftarrow sensed(L_1,t), uBr(t,B), knows(L_2,t-1,t,B), complement(L_1,L_2) \quad \text{(D.17a)}$$
$$apply(EP,t-1,t,B) \leftarrow hasEP(A,EP), exoHappened(A,t-1,B) \quad \text{(D.17b)}$$

The volatile part $Q[t]$ is described by equations (D.18). It represents rules for plan verification (FO5) – (Γ_verify^o).

$$\leftarrow not\ allWGAchieved(t) \quad \text{(D.18a)}$$
$$\leftarrow notSG(t,B), uBr(t,B) \quad \text{(D.18b)}$$
$$\text{(D.18c)}$$

D.4. Problem Specification for Online Planning Use Case

The following Listing D.5 is the original PDDL-like input for the use case depicted and discussed in Section 6.2.

Listing D.5: Domain specification for use case in Section 6.2

```
1
2  ;Types
3    (:types
4      Door
5      Room
6      Agent
7      Person - Agent
8      Robot - Agent)
9
10   ;Fluents
11   (:predicates
12     (hasDoor ?r - Room ?d - Door)
13     (connected ?r1 - Room ?r2 - Room)
14     (inRoom ?ag - Agent ?roo - Room)
15     (open ?d - Door)
16     (abnormal_drive ?r - Robot))
17
18   ;Objects
19   (:objects
20     corr1,bed,couch,office, bath,kit - Room
21     d1,d2,d4 - Door
22     rolland1, rolland2 - Robot
23     fred - Person)
24
25   ; Room layout:
26   ; ; ;-------|----------|--------|
27   ; ; ; (fred)|          |    bed |
28   ; ; ; couch d2  corr1  d1       |
29   ; ; ;  (r1) |---d4-----|  (r2)  |
30   ; ; ;  kit  | bath     | office |
31   ; ; ;-------|-------------------|
32
33   ; Initial knowledge:
34   (:init
35     inRoom(rolland1,kit)
36     inRoom(rolland2,office)
37     inRoom(fred,couch)
38     hasDoor(corr1, d1)
39     hasDoor(corr1, d2)
40     hasDoor(corr1, d4)
```

```
41      hasDoor(bed, d1)
42      hasDoor(couch, d2)
43      hasDoor(bath, d4)
44      connected(office, bed)
45      connected(bed, office)
46      connected(kit, couch)
47      connected(couch, kit)
48
49      !open(d1)
50      !open(d2)
51      !open(d4)
52      (oneof
53        inRoom(rolland1,corr1)
54        inRoom(rolland1,bed)
55        inRoom(rolland1,office)
56        inRoom(rolland1,couch)
57        inRoom(rolland1,kit)
58        inRoom(rolland1,bath))
59
60      (oneof
61        inRoom(rolland2,corr1)
62        inRoom(rolland2,bed)
63        inRoom(rolland2,office)
64        inRoom(rolland2,couch)
65        inRoom(rolland2,kit)
66        inRoom(rolland2,bath))
67
68      (oneof
69        inRoom(fred,corr1)
70        inRoom(fred,bed)
71        inRoom(fred,office)
72        inRoom(fred,couch)
73        inRoom(fred,kit)
74        inRoom(fred,bath)))
75
76    ;Actions
77    (:action openDoor
78      :parameters (?d - Door)
79      :precondition
80      :effect open(?d))
81
82    (:action close_door_exo exogenous
83      :parameters (?d - Door)
84      :precondition
85      :effect !open(?d))
86
87    (:action self_drive_door
88      :parameters (?p - person ?robo - Robot ?door - Door ?from ?to
          - Room)
```

```
89      :precondition (and
90        open(?door)
91        hasDoor(?from, ?door)
92        hasDoor(?to, ?door)
93        inRoom(?p, ?from)
94        !inRoom(?p, ?to)
95        inRoom(?robo, ?from)
96        !inRoom(?robo, ?to))
97      :effect (and
98          inRoom(?robo, ?to)
99          !inRoom(?robo, ?from)
100         inRoom(?p, ?to)
101         !inRoom(?p, ?from)))
102
103   (:action self_drive_direct
104     :parameters (?p - person ?robo - Robot ?from ?to - Room)
105     :precondition (and
106       connected(?from, ?to)
107       inRoom(?p, ?from)
108       !inRoom(?p, ?to)
109       inRoom(?robo, ?from)
110       !inRoom(?robo, ?to))
111     :effect (and
112         inRoom(?robo, ?to)
113         !inRoom(?robo, ?from)
114         inRoom(?p, ?to)
115         !inRoom(?p, ?from)))
116
117   (:action drive_door
118     :parameters (?robo - Robot ?door - Door ?from ?to - Room)
119     :precondition (and
120       open(?door)
121       hasDoor(?from, ?door)
122       hasDoor(?to, ?door)
123       inRoom(?robo, ?from)
124       !inRoom(?robo, ?to))
125     :effect
126       (if !abnormal_drive(?robo) then
127       (and
128         inRoom(?robo, ?to)
129         !inRoom(?robo, ?from))))
130
131   (:action drive_direct
132     :parameters (?robo - Robot ?from ?to - Room)
133     :precondition (and
134       connected(?from, ?to)
135       inRoom(?robo, ?from)
136       !inRoom(?robo, ?to))
137     :effect
```

```
138         (if !abnormal_drive(?robo) then
139         (and
140           inRoom(?robo, ?to)
141           !inRoom(?robo, ?from))))
142
143     (:action senseLocation
144       :parameters (?robo - Robot ?room - Room)
145       :precondition
146       :observe inroom(?robo, ?room))
```

E Dissertation-related Publications

E.1. Publications with Shared Content

The following articles share content with this dissertation.

- Content of Section 3.4 and Chapter 4 as well as the use case in Section 6.1 has been published at the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013) under the title *"Approximate Epistemic Planning with Postdiction as Answer-Set Programming"*. The article has been published at Springer and can be accessed at `http://link.springer.com/chapter/10.1007/978-3-642-40564-8_29`.

  Eppe, M., Bhatt, M., Dylla, F., 2013. Approximate Epistemic Planning with Postdiction as Answer-Set Programming, in: Prodeedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning.

- An extended version of the LPNMR article is published as a technical report in the arXiv repository under the title *"h-approximation: History-Based Approximation of Possible World Semantics as ASP"*. The report can be accessed at `http://arxiv.org/abs/1304.4925`.

  Eppe, M., Bhatt, M., Dylla, F., 2013. h-approximation: History-Based Approximation to Possible World Semantics as ASP. Technical Report. arXiv:1304.4925v1. `arXiv:1304.4925v1`.

- The content of Chapter 5 and the description of the ExpCog system in Section 7.2 has been presented and published at the 11th International Symposium on Logical Formalizations of Commonsense Reasoning (CR 2013) . The article can be accessed at `http://www.commonsense2013.cs.ucy.ac.cy/program.html`.

Eppe, M., Bhatt, M., 2013. Narrative based Postdictive Reasoning for Cognitive Robotics, in: 11th International Symposium on Logical Formalizations of Commonsense Reasoning.

In all papers, the co-authors Mehul Bhatt and Frank Dylla had an advising role and were mainly responsible for the presentation and the structuring of the articles. Scientific content and results, as well as most of the text in the articles was generated by Manfred Eppe.

E.2. Other publications

The following publications do not share content with this thesis. They are preliminary in the sense that they led the author to the topic and the contributions of this thesis.

Eppe, M., Dietrich, D., 2012. Interleaving Planning and Plan Execution with Incomplete Knowledge in the Event Calculus, in: Proceedings of the 6th Starting Artificial Intelligence Research Symposium.

Eppe, M., Dylla, F., 2013. Towards Epistemic Planning Agents, in: Proceedings of the 5th International Conference on Agents and Artificial Intelligence.

Manfred Eppe generated most content and results in both articles. The co-authors Frank Dylla and Dominik Dietrich had an advising role.

List of Figures

1.1. State transition in the case of complete knowledge 3
1.2. Epistemic state transitions with knowledge loss 3
1.3. Epistemic state transitions with sensing 4
1.4. Epistemic state transitions with postdiction 4
1.5. Possible worlds model for epistemic state transitions 7
1.6. 0-approximation model for epistemic state transitions 8
1.7. h-approximation model for temporal epistemic state transitions 13
1.8. The autonomous wheelchair *Rolland* operating in the Smart Home *BAALL* 15

3.1. $\mathcal{A}_k$ semantics 69

5.1. Architecture of the online planning framework 90

6.1. Use case 1: abnormality detection in the Smart Home *BAALL* 105
6.2. Abnormality detection as postdiction with *h-approximation* 106
6.3. Use case 2: interleaving planning, plan execution and abductive explanation 109

List of Tables

2.1. Common modal axiom schemata and their corresponding frame conditions 41
2.2. Common Modal Logic systems and their axiomatization 41
2.3. Survey on epistemic action theories 45

3.1. Relation between $\mathcal{A}_k$ syntax and our PDDL dialect 67

4.1. Relation between ASP formalization of $\mathcal{HPX}$ and its operational semantics 87

6.1. Computation time required to solve online planning tasks 110
6.2. Comparison of different planners for benchmark problems from literature 111

A.1. Relation between Logic Program and operational semantics 132

List of Theorems and Lemmata

Theorem 2.1 – Least Model of a Logic Program 25
Theorem 2.2 – Least fixpoint of a positive Logic Program 26
Theorem 2.3 – Complexity for positive Logic Programs 33
Theorem 2.4 – Complexity for normal Logic Programs 33
Theorem 2.5 – Complexity for normal Logic Programs with optimization . . 33

Theorem 3.1 – Complexity of the $\mathcal{HPX}$ planning problem 66
Theorem 3.2 – Equivalence of $\mathcal{A}_k$ and $\mathcal{A}_k^{TQS}$ for $t = n$ 72
Theorem 3.3 – Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k{}^{TQS}$ 72

Theorem 4.1 – Soundness of ASP Formalization of $\mathcal{HPX}$ 86

Lemma A.1 – Soundness lemma in general form 132
Lemma A.2 – Soundness for knowledge atoms for single state transitions . . 134
Lemma A.3 – Soundness for inertia in induction step 150
Lemma A.4 – Knowledge does not exist in new branches 154
Lemma A.5 – Knowledge does not exist in unused branches 154
Lemma A.6 – Knowledge generation in an $\mathcal{HPX}$-Logic Program 157
Lemma A.7 – Soundness of application of effect propositions 159
Lemma A.8 – Branching of application of effect propositions 166
Lemma A.9 – Actions do not occur in new branches 166
Lemma A.10 – Soundness for sensing results 167
Lemma A.11 – Only one sensing result per branch 169
Lemma A.12 – Soundness for auxiliary predicates 170
Lemma A.13 – Current Step Number 170

Lemma B.1 – Solving the projection problem is polynomial 171
Lemma B.2 – Applying the extended transition function is polynomial 172
Lemma B.3 – Applying the transition function is polynomial 172
Lemma B.4 – Applying $evalOnce$ (B.2) is polynomial 173
Lemma B.5 – Function $evalOnce$ is called constantly often by $eval$ 173
Lemma B.6 – Maximal size of knowledge history 173
Lemma B.7 – Re-evaluation is monotonic 175
Lemma B.8 – Knowledge-persistence for CCP 176
Lemma B.9 – Knowledge producing mechanisms for single state transitions . 177

Lemma C.1 – Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$ for sequences of actions 180
Lemma C.2 – Soundness of the initial state 180
Lemma C.3 – Soundness of $\mathcal{HPX}$ wrt. $\mathcal{A}_k^{TQS}$ for single state transitions . . . 181
Lemma C.4 – Step number for sequences of actions 197
Lemma C.5 – Knowledge persistence in re-evaluated c-states 197
Lemma C.6 – Re-evaluation does not affect knowledge about the presence . . 198

List of Definitions

Definition 2.1 – The projection problem for operational action theories 20
Definition 2.2 – The planning problem for operational action theories 21
Definition 2.3 – The projection problem for model-theoretic action theories . 22
Definition 2.4 – The planning problem for model-theoretic action theories . . 22
Definition 2.5 – Circumscription 23
Definition 2.6 – Minimal Model of a Logic Program 25
Definition 2.7 – The Gelfond Lischitz Reduct 28
Definition 2.8 – Stable Model 28
Definition 2.9 – Logic program module (Oikarinen and Janhunen, 2006) . . . 34
Definition 2.10 – Positive dependency graph of a Logic Program 35
Definition 2.11 – Join of LP modules (Gebser et al., 2011a) 35
Definition 2.12 – Answer Sets of Modules (Gebser et al., 2011a) 35
Definition 2.13 – Projecting rules onto atoms (Gebser et al., 2008) 36
Definition 2.14 – Relating non-ground LPs to ground modules (Gebser et al., 2008) 37
Definition 2.15 – Online Progression (Gebser et al., 2011a) 38
Definition 2.16 – Modularity of Online Progressions (Gebser et al., 2011a) . . 39

Definition 3.1 – Effect history ϵ 54
Definition 3.2 – Initial h-state $\mathfrak{h}_0$ 54
Definition 3.3 – Executability of actions 56
Definition 3.4 – Intermediate h-states 57
Definition 3.5 – Concurrent Conditional Plan 63
Definition 3.6 – Re-evaluated initial k-state 70
Definition 3.7 – Re-evaluated c-states 70

Definition 4.1 – Notation to relate ASP implementation with $\mathcal{HPX}$ semantics 85
Definition 5.1 – Static Relations 96

List of Symbols

Symbol	Name	Description	Definition
Chapter 3			
a	Action	An action $a \in \mathcal{A}$ is a triple $\langle \mathcal{EP}^a, \mathcal{EXC}^a, \mathcal{KP}^a \rangle$.	Section 3.2
f, l	Fluent / literal	A fluent f is a world property and a literal l is a fluent paired with a (boolean) value. We use l^c to denote condition liteals of an effect proposition, l^e to denote effect literals of an effect propositions and f^s to denote a fluent which is sensed by a sensing action.	Section 3.1
$\mathcal{D}$	Planning domain	Denotes a planning domain $\mathcal{D} = \langle \mathcal{VP}, \mathcal{ISC}, \mathcal{A}, \mathcal{G} \rangle$	Section 3.2
$\mathcal{A}$	Set of domain actions	The set of all actions in a planning domain $\mathcal{D}$. Also denoted $\mathcal{A}_{\mathcal{D}}$.	Section 3.2
$\mathcal{VP}$	Value proposition	Denotes a set of literals which are known to hold in the initial state.	Section 3.1, (3.1a)

$\mathcal{ISC}$	Initial state constraint	Denotes a set of initial state constraints $\mathcal{C}$, where $\mathcal{C}$ is a set of literals of which exactly one holds in the initial state.	Section 3.1, (3.1b)
$\mathcal{EXC}^a$	Executability condition of action a	Denotes a set of literals which an agent must know to execute an action.	Section 3.1, (3.1e)
$\mathcal{EP}^a$	Effect propositions of action a	Denotes a set of conditional effects of an action a. An effect proposition $ep \in \mathcal{EP}^a$ has condition literals $c(ep) = \{l_1^c, \ldots, l_k^c\}$ and an effect literal $e(ep) = l^e$.	Section 3.1, (3.1c)
$\mathcal{KP}^a$	Knowledge proposition of action a	Denotes a fluent f^s which is sensed by the action $\mathcal{KP}^a = f^s$.	Section 3.1, (3.1d)
$\mathcal{G}$	Goal proposition	$\mathcal{G}$ is a pair of weak and strong goals defined in a planning domain $\mathcal{D}$	Section 3.1, (3.1f)
$\mathcal{F}_\mathcal{D}$, $\mathcal{L}_\mathcal{D}$	Domain fluents / domain literals	The set of domain fluents (resp. domain literals) defined by the domain description $\mathcal{D}$.	Section 3.1
$\mathfrak{h}$	h-state	An h-state $\mathfrak{h} = \langle \boldsymbol{\alpha}, \boldsymbol{\kappa} \rangle$ is a "history"-aware knowledge state of an $\mathcal{HPX}$-agent constituted by a knowledge history $\boldsymbol{\kappa}$ and an action history $\boldsymbol{\alpha}$. See also function $\mathfrak{h}(n, b, S)$ (4.5) which maps a Stable Model S to an h-state.	Section 3.2.1
$\tilde{\mathfrak{h}}$	Intermediate h-state	An intermediate h-state is an h-state which is not completely evaluated by the $eval$ function (3.17).	Definition 3.4
$\boldsymbol{\alpha}$	Action history	An action history is a set of pairs of action symbols and time steps. Denotes the occurrence of past actions of an h-state. $\boldsymbol{\alpha}(\mathfrak{h})$ denotes the action history of h-state $\mathfrak{h}$	Section 3.2.1

$\boldsymbol{\epsilon}$	Effect history	An effect history is a set of pairs of effect proposition symbols and time steps. Represents which effect propositions have been applied in the past. $\boldsymbol{\epsilon}(\mathfrak{h})$ denotes the effect history of h-state $\mathfrak{h}$	Section 3.2.1, Definition 3.1
$\boldsymbol{\kappa}$	Knowledge history	A knowledge history is a set of pairs of fluent symbols and time steps. Represents temporal knowledge about the world. $\boldsymbol{\kappa}(\mathfrak{h})$ denotes the knowledge history of h-state $\mathfrak{h}$	Section 3.2.1
Ψ	$\mathcal{HPX}$-transition function	Ψ maps a set of actions $\mathcal{A}$ and an h-state $\mathfrak{h}$ to a set of h-states $\mathfrak{h}'$.	Section 3.2.5, (3.7)
p	Plan	A plan p is a syntactic construct which defines a course of actions. p may be concurrent and conditional.	Definition 3.5
$\widehat{\Psi}$	Extended transition function	$\widehat{\Psi}$ maps a concurrent conditional plan and an initial h-state $\mathfrak{h}_0$ to a set of h-states $\mathfrak{h}'$.	Section 3.2.10, (3.18)
δ, δ_n^t	c-state	A c-state $\delta = \langle u, \Sigma \rangle$ is a combined state which is constituted by a state u which reflects an assumed real world and a k-state Σ which represents an agent's knowledge about the world given that u is the real world. A re-evaluated c-state δ_n^t is a c-state with a re-evaluated k-state Σ_n^t.	δ: Section 3.4.2, δ_n^t: Definition 3.7, (3.28)
u, s	State	A state s is a set of fluents f. If $f \in s$ then f holds in s.	Section 3.4.2

Σ, Σ_n^t	k-state	In the $\mathcal{A}_k$-semantics, a k-state Σ is a set of states which represents the knowledge of an agent. A *re-evaluated* k-state Σ_n^t represents the knowledge of an agent at step n about hos the world is at step t.	Σ: Section 3.4.2, Σ_n^t: Definition 3.7, (3.29)
Φ	$\mathcal{A}_k$-transition function	Ψ maps an action a and a c-state δ to a c-state δ'.	Section 3.4.2, (3.24), (3.26)
Chapter 4			
$knows(l,t,n,b)$	Knowledge predicate	Denotes that at step n in branch b it is known that l holds (or did hold) at step t.	Section 4.1
$occ(a,n,b)$	Action occurrence predicate	Denotes that action a occurs at step n in branch b.	Section 4.1
$apply(ep,n,b)$	Effect proposition application predicate	Denotes that an effect proposition ep is applied at step n in branch b.	Section 4.1
$sRes(l,n,b,b')$	Sensing result predicate	Denotes that the literal l is sensed at step n in branch b, such that it will hold in the child branch b'.	Section 4.1
$uBr(n,b)$	Used branch predicate	Denotes that branch b is a valid branch at step n. Actions can only be executed if a branch is valid.	Section 4.1
$LP(\mathcal{D})$	$\mathcal{HPX}$-Logic Program of a domain $\mathcal{D}$	Is a conjunction of the domain independent theory Γ_{hpx} and the domain specific theory Γ_{world}.	Section 4.2, (4.1)
Γ_{hpx}	Domain independent part of an $\mathcal{HPX}$-Logic Program	$\Gamma_{hpx} = \Gamma_{aux} \cup \Gamma_{in} \cup \Gamma_{sen} \cup \Gamma_{infer} \cup \Gamma_{conc} \cup \Gamma_{verify} \cup \Gamma_{plan}$ is a conjunction of sets of Logic Programming rules which constitute the domain independent part of an $\mathcal{HPX}$-Logic Program.	Section 4.2, (4.1)

Γ_{world}	Domain specific part of an $\mathcal{HPX}$-Logic Program	$\Gamma_{world} = \Gamma_{init} \cup \Gamma_{act} \cup \Gamma_{goal}$ is a conjunction of sets of Logic Programming rules which are generated by translation rules (T1) – (T8) and which constitute the domain specific part of an $\mathcal{HPX}$-Logic Program.	Section 4.2, (4.1)
$kCause(l, t, n, b)$	Knowledge by causation predicate	Denotes that at step n in branch b it is known by causation that l holds (or did hold) at step t.	Section 4.3, (T6a)
$kPosPost(l, t, n, b)$	Knowledge by positive postdiction predicate	Denotes that at step n in branch b it is known by positive postdiction that l holds (or did hold) at step t.	Section 4.3, (T6b)
$kNegPost(l, t, n, b)$	Knowledge by negative postdiction predicate	Denotes that at step n in branch b it is known by negative postdiction that l holds (or did hold) at step t.	Section 4.3, (T6c)
$kNotSet(l, t, n, b)$	Inertia predicate	Denotes that at step n in branch b it is known that l is not set at step t, respectively that $\bar{l}$ is inertial at step t.	Section 4.4, (F3)
`maxS`, `maxB`	Plan size constants	`maxS` is a constant which restricts the maximal plan length and `maxB` is a constant which restricts the maximal plan width.	Section 4.5
$\mathfrak{h}(n, b, S)$	h-state function	$\mathfrak{h}(n, b, S) = \langle \boldsymbol{\alpha}(n, b, S), \boldsymbol{\kappa}(n, b, S) \rangle$ is a function that relates a Stable Model S to an h-state, where $\boldsymbol{\alpha}(n, b, S)$ extracts the action occurrence predicates from a Stable Model and $\boldsymbol{\kappa}(n, b, S)$ extracts the knowledge predicates.	Section 4.5
$S_{\mathcal{D}}^{\boldsymbol{P}}$	Stable Model of $\mathcal{HPX}$-LP	$S_{\mathcal{D}}^{\boldsymbol{P}}$ is a Stable Model of $LP(\mathcal{D}) \cup \boldsymbol{P}$ where $\boldsymbol{P}$ is a set of $occ(a, n, b)$ atoms which represent a plan.	Section 4.5, Definition 4.1

$\boldsymbol{A}_{n,b}$	Action occurrence at n, b	$\boldsymbol{A}_{n,b} = \{a \mid occ(a, n, b) \in S_{\mathcal{D}}^{\boldsymbol{P}}\}$ is a set of actions applied at a transition tree node with the "coordinates" $\langle n, b \rangle$.	Section 4.5, Definition 4.1
Chapter 5			
$LP(\mathcal{N})$	Execution narrative	$LP(\mathcal{N})$ is a set of $exec/2$ and $sensed/2$ atoms which reflect which actions were executed and which sensing results were obtained.	Section 5.1
$sensed(l, t)$	Sensing predicate	Denotes that sensing revealed that l holds at step t.	Section 5.2.2
$exec(a, t)$	Execution predicate	Denotes that action a was executed at step t.	Section 5.2.2

Bibliography

Baral, C., Kreinovich, V., Trejo, R., 2000. Computational complexity of planning and approximate planning in the presence of incompleteness. Artificial Intelligence 122, 241–267.

Bengson, J., Moffett, M.A. (Eds.), 2012. Knowing How: Essays on Knowledge, Mind, and Action. Oxford University Press, USA.

Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2001. MBP : a Model Based Planner, in: Proceedings of the International Joint Conference on Artificial Intelligence.

Blackburn, P., de Rijke, M., Venema, Y., 2001. Modal Logic. Cambridge University Press.

Brenner, M., Nebel, B., 2009. Continual planning and acting in dynamic multiagent environments. Autonomous Agents and Multi-Agent Systems 19, 297–331.

Cimatti, A., Pistore, M., Roveri, M., Traverso, P., 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence .

Demolombe, R., del Pilar Pozos Parra, M., 2000. A simple and tractable extension of situation calculus to epistemic logic, in: International Symposium on Foundations of Intelligent Systems.

van Ditmarsch, H., van der Hoek, W., Kooi, B., 2007. Dynamic Epistemic Logic. Springer.

Doherty, P., 1994. Reasoning about Action and Change using Occlusion, in: Proceedings of the 11th European Conference on Artificial Intelligence, pp. 401–405.

Eiter, T., Ianni, G., Krennwallner, T., 2009. Answer Set Programming - A Primer, in: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.C., Schmidt, R.A. (Eds.), Reasoning Web. Semantic Technologies for Information Systems. Springer-Verlag, Berlin, Heidelberg, pp. 40–110.

Eppe, M., Bhatt, M., 2013. Narrative based Postdictive Reasoning for Cognitive Robotics, in: 11th International Symposium on Logical Formalizations of Commonsense Reasoning.

Eppe, M., Bhatt, M., Dylla, F., 2013a. Approximate Epistemic Planning with Postdiction as Answer-Set Programming, in: Prodeedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning.

Eppe, M., Bhatt, M., Dylla, F., 2013b. h-approximation: History-Based Approximation to Possible World Semantics as ASP. Technical Report. arXiv:1304.4925v1. arXiv:1304.4925v1.

Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y., 1995. Reasoning About Knowledge. MIT Press.

Ferrier, J.F., 1854. Institutes of Metaphysics: The Theory of Knowing and Being. W. Blackwood, Edinburgh.

Fikes, R., Nilsson, N., 1972. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial intelligence .

Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T., 2012a. Stream Reasoning with Answer Set Programming: Preliminary Report, in: International Conference on Principles of Knowledge Representation and Reasoning.

Gebser, M., Grote, T., Kaminski, R., Schaub, T., 2011a. Reactive Answer Set Programming, in: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S., 2008. Engineering an Incremental ASP Solver, in: International Converence on Logic Programming.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2012b. Answer Set Solving in Practice. Morgan and Claypool.

Gebser, M., Kaminski, R., König, A., Schaub, T., 2011b. Advances in gringo series 3, in: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning.

Gelfond, M., 1994. Logic programming and reasoning with incomplete information. Annals of Mathematics and Artificial Intelligence .

Gelfond, M., Lifschitz, V., 1988. The Stable Model Semantics for Logic Programming, in: Proceedings of the International Conference on Logic Programming (ICLP).

Gelfond, M., Lifschitz, V., 1991. Classical negation in logic programs and disjunctive databases. New generation computing .

Gelfond, M., Lifschitz, V., 1993. Representing action and change by logic programs. The Journal of Logic Programming 17, 301–321.

Gelfond, M., Lifschitz, V., 1998. Action languages. Electronic Transactions on Artificial Intelligence 3, 1–23.

de Giacomo, G., Levesque, H.J., 1998. An Incremental Interpreter for High-Level Programs with Sensing, in: Working Notes of the 1998 AAAI Fall Symposium on Cognitive Robotics.

Giunchiglia, F., Lee, J., Lifschitz, V., McCain, N., Turner, H., 2004. Nonmonotonic Causal Theories. Artificial Intelligence 153, 49 –104.

Giunchiglia, F., Lifschitz, V., 1998. An Action Language based on Causal Explanation: A Preliminary Report, in: AAAI Conference on Artificial Intelligence.

Goldblatt, R.I., 2003. Mathematical Modal Logic: A View of its Evolution. Journal of Applied Logic 1, 309—-392.

Hanks, S., McDermott, D., 1987. Nonmonotonic logic and temporal projection. Artificial Intelligence 33, 379 – 412.

Heisenberg, W., 1927. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. Zeitschrift für Physik 43, 172–198.

Hintikka, J., 1962. Logic and Belief: An Introduction to the Logic of the two Notions. Cornell University Press, Ithaca.

Hoffmann, J., Brafman, R.I., 2005. Contingent planning via heuristic forward search with implicit belief states, in: Proceedings of the International Conference on Automated Planning and Scheduling.

Hölldobler, S., Schneeberger, J., 1990. A new deductive approach to planning. New Generation Computing 8, 225–244.

Kahramanogullari, O., Thielscher, M., 2003. A Formal Assessment Result for Fluent Calculus Using the Action Description Language Ak, in: German Conference on Artificial Intelligence.

Kowalski, R., 1974. Predicate Logic as Programming Language, in: Proceedings of International Federation for Information Processing, pp. 569– 574.

Kowalski, R., Sergot, M., 1986. A Logic-based calculus of events. New generation computing 4, 67–94.

Krieg-Brückner, B., Röfer, T., Shi, H., Gersdorf, B., 2010. Mobility Assistance in the Bremen Ambient Assisted Living Lab. GeroPsych: The Journal of Gerontopsychology and Geriatric Psychiatry 23, 121–130.

Krieg-Brückner, B., Shi, H., Gersdorf, B., Döhle, M., Röfer, T., 2012. Context-Sensitive Spatial Interaction and Ambient Control, in: Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives.

Kripke, S., 1963. Semantical Considerations on Modal Logic. Acta Philisophica Fennica 16, 83–94.

Kvarnström, J., 2005. TALplanner and other extensions to temporal action logic. Ph.D. thesis. Lingköpings Universitet.

Lakemeyer, G., Levesque, H.J., 1998. AOL: a logic of acting, sensing, knowing, and only knowing, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, pp. 316 –327.

Lee, J., Palla, R., 2009. System f2lp – Computing Answer Sets of First-Order Formulas, in: Logic Programming and Nonmonotonic Reasoning, pp. 515–521.

Levesque, H.J., 1990. All I know: A study in autoepistemic Logic. Artificial Intelligence 43, 263–309.

Lifschitz, V., 1994. Circumscription, in: Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3. Oxford University Press, pp. 297 – 343.

Liu, Y., Levesque, H.J., 2005. Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions, in: International Joint Conference on Artificial Intelligence.

Lobo, J., Mendez, G., Taylor, S., 2001. Knowledge and the Action Description Language A. Theory and Practice of Logic Programming 1, 129–184.

Ma, J., Miller, R., Morgenstern, L., Patkos, T., 2013. An Epistemic Event Calculus for ASP-based Reasoning About Knowledge of the Past, Present and Future, in: International Conference on Logic for Programming, Artificial Intelligence and Reasoning.

Mandel, C., Huebner, K., Vierhuff, T., 2005. Towards an Autonomous Wheelchair: Cognitive Aspects in Service Robotics, in: Towards Autonomous Robotic Systems, pp. 165–172.

McCain, N., Turner, H., 1995. A causal theory of ramifications and qualifications. International Joint Conference on Artificial Intelligence .

McCarthy, J., 1959. Programs with Common Sense, in: Proceedings of the Teddington Conference on the Mechanization of Thought Processes, pp. 75–91.

McCarthy, J., 1963. Situations, Actions and Causal Laws. Technical Report July. Stanford Artificial Intelligence Project.

McCarthy, J., 1980. Circumscription - A form of non-monotonic reasoning. Artificial Intelligence 13, 27–39.

McCarthy, J., 1998. Elaboration tolerance, in: International Symposium on Logical Formalizations of Commonsense Reasoning.

McCarthy, J., Hayes, P., 1969. Some philosophical problems from the standpoint of artificial intelligence. Machine intelligence 4, 463–502.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Wilkins, D., 1998. PDDL - The Planning Domain Definition Language. Technical Report. Yale Center for Computational Vision and Control.

Miller, R., Morgenstern, L., Patkos, T., 2013. Reasoning About Knowledge and Action in an Epistemic Event Calculus, in: International Symposium on Logical Formalizations of Commonsense Reasoning.

Moore, R.C., 1985. A formal theory of knowledge and action, in: Hobbs, J., Moore, R.C. (Eds.), Formal theories of the commonsense world. Ablex, Norwood, NJ.

Oikarinen, E., Janhunen, T., 2006. Modular equivalence for normal logic programs, in: Proceedings of the Europeam Conference on Artificial Intelligence.

Patkos, T., 2010. A Formal Theory for Reasoning About Action , Knowledge and Time. Ph.D. thesis. University of Crete - Heraklion Greece.

Patkos, T., Plexousakis, D., 2009. Reasoning with Knowledge , Action and Time in Dynamic and Uncertain Domains, in: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 885–890.

Patkos, T., Plexousakis, D., 2012. Epistemic and Causal Commonsense Reasoning in Partially Observable Dynamic Domains - From Sensors to Concepts, in: Bridges between the Methodological and Practical Work of the Robotics and Cognitive Systems Communities. Springer.

Pednault, E.P.D., 1994. ADL and the State-Transition Model of Action. Journal of Logic and Computation 4, 467–512.

Petrick, R.P., Bacchus, F., 2004. Extending the knowledge-based approach to planning with incomplete information and sensing, in: Proceedings of the International Conference on Automated Planning and Scheduling.

Przymusinski, T., 1987. On the declarative semantics of stratified deductive databases and logic programs, in: Minker, J. (Ed.), Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, pp. 193 – 216.

Reiter, R., 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression, in: Lifschitz, V. (Ed.), Artificial intelligence and mathematical theory of computation. Academic Press Professional, Inc., pp. 359–380.

Röfer, T., Mandel, C., Laue, T., 2009. Controlling an automated wheelchair via joystick/head-joystick supported by smart driving assistance, in: IEEE International Conference on Rehabilitation Robotics, Ieee. pp. 743–748.

Sandewall, E., 1994. Features and fluents: The representation of knowledge about dynamical systems. Clarendon Press, Oxford.

Sardina, S., de Giacomo, G., Lespérance, Y., Levesque, H.J., 2004. On the semantics of deliberation in IndiGolog – from theory to implementation. Annals of Mathematics and Artificial Intelligence , 259–299.

Scherl, R.B., Levesque, H.J., 2003. Knowledge, action, and the frame problem. Artificial Intelligence 144, 1–39.

Son, T.C., Baral, C., 2001. Formalizing sensing actions - A transition function based approach. Artificial Intelligence 125, 19–91.

Suchan, J., Bhatt, M., 2013. The ExpCog Framework: High-Level Spatial Control and Planning for Cognitive Robotics. Bridges between the Methodological and Practical Work of the Robotics and Cognitive Systems Communities: From Sensors to Concepts .

Thiebaux, S., Hoffmann, J., Nebel, B., 2003. In Defense of PDDL Axioms, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning.

Thielscher, M., 1998. Introduction To The Fluent Calculus. Linköping Electronic Articles in Computer and Information Science 3.

Thielscher, M., 2000. Representing the knowledge of a robot, in: International Conference on Principles of Knowledge Representation and Reasoning.

Thielscher, M., 2001. The concurrent, continuous fluent calculus. Studia Logica 67, 315–331.

Thielscher, M., 2005. FLUX : A Logic Programming Method for Reasoning Agents. Theory and Practice of Logic Programming 5.

To, S.T., 2011. On the impact of belief state representation in planning under uncertainty, in: Proceedings of the International Joint Conference on Artificial Intelligence.

To, S.T., 2012. A New Approach to Contingent Planning Using A Disjunctive Representation in AND / OR forward Search with Novel Pruning Techniques. Journal of Artificial Intelligence Research .

Tu, P.H., Son, T.C., Baral, C., 2007. Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. Theory and Practice of Logic Programming 7, 377–450.

Turner, H., 1999. A Logic of Universal Causation. Artificial Intelligence 113, 87–123.

Vlaeminck, H., Vennekens, J., Denecker, M., 2012. A general representation and approximate inference algorithm for sensing actions, in: Advances in Artificial Intelligence - Australasian Joint Conference.

Weld, D.S., Anderson, C.R., Smith, D.E., 1998. Extending Graphplan to handle uncertainty and sensing actions, in: Proceedings of the Inernational Conference of the Association for the Advancement of Artificial Intelligence.

Williamson, T., 2002. Knowledge and its Limits. Oxford University Press, USA.

Winograd, T., 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical Report. Massachusetts Institute of Technology.