Planning and Verification in the Agent Language Golog

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Jens Claßen

aus Willich

Berichter:
Prof. Gerhard Lakemeyer, Ph.D.
Prof. Dr.-Ing. Franz Baader

Tag der mündlichen Prüfung: 26. April 2013

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.
Acknowledgements

First and foremost, I would like to thank my supervisor Gerhard Lakemeyer for giving me the opportunity for this thesis and his invaluable guidance during the (long) time of its preparation. Without his support and patience – together with the right degree of impatience at the right time – this project would never have seen completion. I am also grateful to Franz Baader for assuming the role of the second reviewer of the thesis, as well as Jürgen Giesl and Thomas Seidl for agreeing to act as additional examiners at my defense.

Furthermore, I am indebted to many people with whom I have worked over the last years. I want to express my gratitude towards my colleagues at the Knowledge-Based Systems Group, in particular (in order of appearance) Gero Iwan, Günter Gans, Vazha Amiranashvili, Alexander Ferrein, Stefan Schiffer, Daniel Beck, Vaishak Belle, Tim Niemüller, Christoph Schwering, and Martin Liebenberg.

Moreover, I would like to thank my previous office mates Satish Srirama and Andreas Lorenz as well as the people at the Information Systems and Databases Group, especially Sandra Geisler, Daniele Glöckner, Gabriele Hoeppermanns, Jessica Huster, Matthias Jarke, Tatiana Liberzon, Reinhard Linde, Dominik Lübbers, Claudia Puhl, Dominik Schmitz, and Christoph Quix. I also enjoyed working together with Felix Wolf as well as Renate Eschenbach-Thomas and the team of the Computer Science Library.

Many more people supported or contributed to this work in various ways. I thank my thesis students (in chronological order) Dominique Ziegelmayer, Yuxiao Hu, Jiyoung Han, Bernhard Ern, Marius Grysla, Ziquan Zhu, and Martin Liebenberg, as well as my student assistants Viktor Engelmann, Thomas Ströder, and Niklas Hoppe. I benefited a lot from discussions with Alfredo Gabaldon, Yves Lespérance, Hector Levesque and Yongmei Liu during their research visits in Aachen. I also appreciate the fruitful exchanges I could have with Tim Capes, Christian Fritz, Yilan Gu, Ryan Kelly, Sebastian Sardiña, and Stavros Vassos.

Next, I want to thank our collaboration partners in the LogWiss research network as well as the Hybris research unit, in particular Franz Baader, Conrad Drescher, Patrick Eyerich, Hongkai Liu, Bernhard Nebel, Gabriele Röger, Stephan Schiffer, Michael Thielscher, and Benjamin Zarrieß. Financial support was gratefully received from the German Research Foundation (DFG).

Many thanks go to the people at Scientific Computers and m3connect for providing me employment and a great working atmosphere during the final phase of writing this thesis. I am further grateful to my friends and family for all their support and encouragement.

Finally, I owe a very special terimakasih to my beloved wife Evaline for her continuous love, care and understanding in all those years. This thesis is dedicated to her.
Abstract

The action programming language GOLOG has proven to be a useful means for the high-level control of autonomous agents such as mobile robots. It is based on the Situation Calculus, a dialect of classical first-order logic, that is used to encode dynamic domains through logical axioms. Perhaps the greatest advantage of GOLOG is that a user can write programs which constrain the search for an executable plan in a flexible manner.

However, when general planning is needed, GOLOG supports this only in principle, but does not measure up with state-of-the-art planners, most of which are based on the plan language PDDL. On the other hand, planning formalisms and systems lack the expressiveness of GOLOG that make it suited for realistic scenarios of agents with partial world knowledge acting in dynamic environments. We therefore propose an integration of GOLOG and planning where planning subtasks encountered during the execution of a GOLOG program are referred to a PDDL planner, thus combining GOLOG’s expressiveness with the efficiency of modern planners. The theoretical justification for such an embedding is provided in the form of relating state updates in PDDL to the progression of a certain form of theories of the modal Situation Calculus variant $\mathcal{ES}$. We complement these results with an empirical evaluation that shows that equipping GOLOG with a PDDL planner indeed pays off in terms of the runtime performance.

Moreover, before deploying a GOLOG program onto a robot, it is often desirable to verify that certain requirements are met, typical examples including safety, liveness and fairness conditions. Since autonomous robots typically perform open-ended tasks, the corresponding control programs are often non-terminating. Analyzing such programs so far requires manual, meta-theoretic arguments involving complex fixpoint constructions, which is tedious and error-prone. In this thesis, we propose an extension to $\mathcal{ES}$ that includes new modal operators to express temporal properties of GOLOG programs. We then provide algorithms for the automated verification of such properties, relying on a newly introduced graph representation for GOLOG programs which enables a systematic exploration of the state space. Similar to other forms of reasoning in the Situation Calculus, our verification methods ultimately reduce to classical first-order theorem proving.
Zusammenfassung


# Contents

1 Introduction .......................................................... 1
  1.1 Motivation .......................................................... 1
    1.1.1 Planning ...................................................... 4
    1.1.2 Verification .................................................. 6
  1.2 Objectives and Contributions ...................................... 7
  1.3 Outline ........................................................... 8

2 Related Work ................................................................ 11
  2.1 Action Formalisms .................................................... 11
  2.2 Planning ............................................................... 15
  2.3 Verification ............................................................ 19

3 Logical Foundations ..................................................... 23
  3.1 A Modal First-Order Logic for Dynamic Domains ............... 23
    3.1.1 First-Order Representation and Reasoning .................. 23
    3.1.2 Standard Names ............................................... 25
    3.1.3 Fluents, Actions and Situations ............................... 27
  3.2 The Logic $\mathcal{L}S$ ............................................... 29
    3.2.1 Formal Syntax ................................................. 30
    3.2.2 Formal Semantics ............................................. 33
    3.2.3 Properties ...................................................... 35
    3.2.4 A Remark on Numbers ........................................ 43
    3.2.5 Second-Order Quantification ................................ 47
  3.3 Basic Action Theories ............................................... 49
    3.3.1 The Qualification Problem ..................................... 50
    3.3.2 The Frame Problem ............................................ 52
    3.3.3 Formal Definition ............................................. 56
  3.4 Projection ........................................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Regression</td>
<td>61</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Progression</td>
<td>67</td>
</tr>
<tr>
<td>3.5</td>
<td>Knowledge and Sensing</td>
<td>77</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Formal Definition</td>
<td>78</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Regression</td>
<td>81</td>
</tr>
<tr>
<td>3.5.3</td>
<td>The Representation Theorem</td>
<td>83</td>
</tr>
<tr>
<td>3.6</td>
<td>GOLOG</td>
<td>86</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Simple GOLOG</td>
<td>88</td>
</tr>
<tr>
<td>3.6.2</td>
<td>CONGOLOG</td>
<td>89</td>
</tr>
<tr>
<td>3.6.3</td>
<td>INDI GOLOG</td>
<td>90</td>
</tr>
<tr>
<td>3.6.4</td>
<td>GOLOG in ES</td>
<td>91</td>
</tr>
<tr>
<td>3.7</td>
<td>A Knowledge-Based Agent</td>
<td>94</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Regression</td>
<td>96</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Progression</td>
<td>101</td>
</tr>
<tr>
<td>3.8</td>
<td>Discussion</td>
<td>107</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Summary</td>
<td>107</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Comparison to Other Formalizations</td>
<td>108</td>
</tr>
<tr>
<td>4</td>
<td>Planning in Golog</td>
<td>111</td>
</tr>
<tr>
<td>4.1</td>
<td>The Planning Domain Definition Language</td>
<td>116</td>
</tr>
<tr>
<td>4.2</td>
<td>The ADL Fragment</td>
<td>122</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Language Definition</td>
<td>124</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Semantics</td>
<td>127</td>
</tr>
<tr>
<td>4.2.3</td>
<td>The Mapping</td>
<td>129</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Correctness</td>
<td>132</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Expressivity</td>
<td>142</td>
</tr>
<tr>
<td>4.3</td>
<td>Evaluation</td>
<td>144</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Embedding PDDL Planning into GOLOG</td>
<td>144</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Experiments and Results</td>
<td>146</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>154</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Summary</td>
<td>154</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Comparison to Related Work</td>
<td>154</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Future Work</td>
<td>156</td>
</tr>
<tr>
<td>5</td>
<td>Golog Verification</td>
<td>161</td>
</tr>
<tr>
<td>5.1</td>
<td>The Logic ES</td>
<td>165</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Situations Tree ................................................................. 29
3.2 GOLOG Constructs ............................................................. 91
3.3 Program Finality .............................................................. 92
3.4 Program Transitions ......................................................... 93

4.1 Control procedure for the daily routine of an office robot ............. 112
4.2 Office robot domain defined in the STRIPS fragment of PDDL ........ 119
4.3 PDDL planning problem instance for the office robot domain ........... 120
4.4 PDDL requirement flags and their meaning .............................. 123
4.5 Logistics: Median runtimes in seconds .................................. 149
4.6 Elevator: Median runtimes in seconds ................................... 151
4.7 Mail Delivery: Median runtimes in seconds ............................... 153

5.1 Example characteristic graphs ............................................. 187
5.2 Characteristic graph for the coffee robot example ...................... 191
5.3 Characteristic graph for the coffee robot example, with preconditions . 201
5.4 Characteristic graph for the program $\delta = (\pi x. \text{Poss}(rC(x))?; rC(x))^*$ 237
5.5 OBDD for the formula $A \lor B \land \neg A$ ..................................... 242
List of Tables

3.1 Syntactical preference among logical operators ........................................ 33
4.1 Evaluation Results Logistics Domain ......................................................... 158
4.2 Evaluation Results Elevator Domain ......................................................... 159
4.3 Evaluation Results Mail Domain ............................................................... 160
Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence (AI) can roughly be defined as the study of intelligent agents, where an agent is an entity that perceives its surroundings through sensors (for example cameras and laser range finders) and acts upon its environment by means of actuators (e.g., robotic arms and wheels). While this broad definition includes a huge range of applications such as computer game characters, automated lawn-mowers and self-driving cars, one of AI’s most appealing visions is to build intelligent, autonomous robots that cooperate and interact with human beings in a natural way, that are capable of dynamically adapting to new situations, and that can learn to fulfill all kinds of desired tasks which they have not explicitly been programmed for.

Although reality is still somewhat behind the promises made in popular culture science fiction [Asi82], substantial progress has been made since the pioneering work on the robot “Shakey” at the Stanford Research Institute in the 1960s and early 1970s [Nil84] and landmark projects such as the museum tour guide robot “Rhino” [BCF+99]. Symptomatic for the growing interest in robotics are the advent of robotics competitions like the ones offered by the Robocup initiative [KAK+97]. Whereas it started as a series of tournaments for Soccer-playing robots, the scope of Robocup has since expanded to include application domains that get closer to the above-mentioned vision of robot companions interacting with humans in domestic and workplace environments. This holds in particular for the “Robocup@Home” league [vdZW07], where the objective is to develop service robotic systems that assist people in their everyday lives within home environments. The robot Caesar [SFL12a, NFBL10] developed by the team of the Aachen Knowledge-Based Systems Group is one example for a robotic system that successfully participated at Robocup@Home competitions.

While nowadays there exist robust approaches for many low-level tasks such as navigation,
collision avoidance and localization, the question of how the high-level control of an autonomous robot in a dynamic, incompletely known environment can be designed to let it behave intelligently is still an issue that is subject of active research. The field of Cognitive Robotics [LL08] is concerned with developing “an understanding of the relationship between the knowledge, the perception, and the action of such a robot” [LR98].

To choose its actions, an intelligent agent has to maintain an internal representation of its own state and that of its environment. Typically, especially in the case of a mobile robot, the agent will only have incomplete knowledge about its surroundings. For example, it may be fully aware of objects and people in the same room, but ignorant of who is currently present in the room next door, unless it takes action and uses its sensors in order to gather the missing information. Furthermore, there may be facts that are not explicitly known to the agent, but that can be deduced from what it knows by means of reasoning. For instance, if fragile objects are known to break when being dropped, and when the agent believes\(^1\) that the vase it is holding is a fragile object, then it can infer that it better not drop the vase. The subarea of AI that studies formalisms for the symbolic representation of knowledge and corresponding inference methods is called Knowledge Representation and Reasoning (KR) [BL04]. Its goal is to encode human knowledge in a way that it can be put to use in an automated system, which is best summarized by the Knowledge Representation Hypothesis [Smi82]:

“Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge.”

That is to say we speak of a knowledge-based system whenever two criteria are met: First, there is something within the system (in the following called its knowledge base) that can easily be understood as propositions, i.e. declarative statements that can be either true or false. Second, it is required that these propositions indeed determine how the system behaves.

The knowledge-based approach is thus especially appealing when the set of tasks the system is supposed to perform is not fixed in advance, in which case there are often more efficient task-specific solutions, but rather open-ended. Clearly this is the case in the intelligent domestic robot scenario. For instance, the robot may at some point learn that books are typically in the library. While there may be no current use for that information, it may become important later on with new tasks imposed on the robot. Moreover, a knowledge-based system can be designed

\(^1\)In this thesis, we use the terms “knowledge” and “belief” as synonyms.
to justify and explain its behaviour. If the robot is told to get a certain book, and we want to know why it starts moving towards the library, then it may answer that this is because it believes the book to be in the library.

Much speaks in favour of taking a logical approach to KR, where the knowledge base is a collection of formulas in some logical language such as propositional logic or first-order logic (FOL). It is certainly possible to represent an agent’s knowledge through natural languages such as English or German, but due to their sheer endless expressiveness, it is very difficult to process statements of this kind by a computer program. Logics however provide a good compromise between human-comprehensibility on the one hand and the availability of powerful reasoning tools on the other hand. With the logical approach, reasoning reduces to logical entailment, and thus to just another form of mechanical symbol manipulation that computers are particularly well suited for.

As opposed to other approaches where knowledge is represented subsymbolically, for example through neural nets [MP43], or in a symbolic, yet non-logical manner, e.g. graphically or procedurally [Min75], one of the main advantages of the logic-based approach lies in the fact that the corresponding formally well-founded, declarative semantics makes the represented knowledge and the behaviour it induces much easier to comprehend. Thus, a logical representation of knowledge not only constitutes the basis for high-level, abstract problem solving, arguably a paramount ingredient for complex intelligent behaviour, but also allows for a formal analysis and verification of the system and its properties.

Regarding the question of what logic is the most appropriate for the purpose of representing an agent’s knowledge, a well accepted opinion is that at least the expressive power of FOL is required. According to Moore [Moo82], the two main reasons for this are that FOL describes the world in terms of objects, their properties and relations, together with the fact that it is capable to express incomplete information so well. Obviously, a domestic robot has to deal with objects, properties and relations and, as argued above, its knowledge is usually incomplete.

Classical FOL by itself is however not enough due to its inherently static nature. Since agents live in dynamic worlds, it is rather necessary to account for changes to the objects’ properties and relations, be they caused by the agent’s actions or exogenously. The Situation Calculus [MH69, Rei01a] is a dialect of first-order logic (with some second-order features) for reasoning about such dynamic domains, and it is probably the best known and most widely studied action logic. It constitutes the basis for the expressive high-level agent programming language GOLOG [LRL+97] in the sense that axioms in the Situation Calculus are used to define the preconditions and effects of actions, and GOLOG allows to compose complex behaviours out of those actions. The latter may contain on the one hand deterministic constructs as known
from imperative programming languages, and on the other hand nondeterministic operators that leave a choice to the system and have to be resolved through search. In addition to the high expressiveness that arises from its Situation Calculus basis, one of the greatest advantages of Golog is that it is thus possible to freely combine programming with planning. Variants of Golog have already been successfully applied to the control of autonomous robots such as the above mentioned Rhino and Caesar.

Despite its popularity, the original Situation Calculus is problematic in certain respects. This becomes particularly apparent in extensions that include epistemic modalities [Rei01b, SL03]. The latter allow formulas to not only express statements about the state of the world, but also the agent’s knowledge and lack thereof, thus allowing for introspective agents that can reason about what they know and what they do not know, and how their knowledge is affected by sensing. However, the corresponding Situation Calculus axiomatization of a possible-world semantics [Hin62, Kri63] is quite involved and to some extends leads to counter-intuitive results. Model-theoretic proofs of meta-theoretical properties moreover tend to be lengthy and tedious. For that reason, Lakemeyer and Levesque introduced the logic $ES$ [LL10, Lak10] which can be viewed as a modal variant of the epistemic Situation Calculus, where belief and action is encoded through modalities whose meaning is defined within the (non-classical) semantics of the logic, rather than axiomatically. This allows for a better readable syntax and much easier proofs, while all the benefits of the Situation Calculus are retained, including its reasoning methods, its first-order expressiveness, and Golog.

Unfortunately, a common theme in KR is that more expressiveness almost always comes at the price of less tractability: In more expressive formalisms, reasoning is computationally much harder, and in the case of FOL even undecidable. For this reason, practical implementations of Golog-based agents typically apply harsh restrictions in order to ensure tractability, thus losing a great deal of expressiveness. Closely related to this issue is the problem of Logical Omniscience [Hin79]: Formally, an agent knows all logical consequences of the beliefs in its knowledge base as well as all tautologies. In practice, the resources of an agent are however bounded, and with limited memory and time, not all inferences can be derived explicitly. Clearly, humans are not logically omniscient as we are not always instantly aware of all implications of the information that we possess. On the contrary, at times it can rather be quite cumbersome if not impossible to come to certain conclusions.

1.1.1 Planning

It was recognized quite early that controlling an agent solely based on logical entailment is infeasible, which gave rise to the AI subarea of Planning, starting with the introduction of the
1.1 Motivation

STRIPS planner [FN71] for the robot Shakey in the early 1970s. Whereas work on action logics such as the Situation Calculus was concerned with formalisms of high expressiveness, planning research focused primarily on efficiency to obtain formalisms and methods that exhibit good computational performance. Despite their common origin, the two fields developed rather independently from each other for over two decades.

One could observe a sudden thrust of progress made in the area of planning in the last one and a half decades. Again, this was in large part due to the upcoming of an international competition by means of which systems by different researchers could be benchmarked and evaluated on a common basis. In particular, the Planning Domain Definition Language PDDL [GHK+98] that is used as the input language for the planners participating at the biennially held International Planning Competition (IPC) has become a de-facto standard for the formulation of planning domains and problems since its introduction in 1998. It extends the very simple STRIPS formalism by various features such as conditional effects, durative actions and concurrency. Today, there is a multitude of planning systems that proved their efficiency at the IPC, and that use a variety of different techniques and approaches such as propositional satisfiability, forward search or partial-order planning, to name but a few.

In many agent scenarios, and especially in the case of an autonomous domestic robot, one often encounters subproblems that are rather combinatorial in nature, such as scheduling currently pending requests, planning a route, or a combination of these two. For example, the robot may have to decide in which order to serve coffee to people, when to get the mail and deliver it to its recipients, and come up with an appropriate path through the building to do so. When it comes to such pure, classical planning tasks, state-of-the-art planning systems usually outperform any current implementation of GOLOG. One may then be tempted to ask why GOLOG is needed at all, for which there are at least two good reasons.

First, the restricted expressiveness of planning languages allow their application only for certain subproblems. In particular, at least in what constitutes the main area of classical, sequential planning, complete information about the problem at hand is required. Therefore, planning is appropriate for tasks like scheduling and routing a robot, but a more expressive logic is still needed to represent and reason about incomplete knowledge, e.g. regarding the current location of people and objects.

Second, in an agent system whose design follows the classical sense-plan-act scheme, planning only constitutes one building block, and there has to be some meta-level control to update the knowledge base according to exogenous events and sensing results, to monitor the execution of the actions in a chosen plan, and to trigger replanning in case a currently executed plan fails. Often, the integration of a planner into such a system is only ad-hoc and meta-theoretic. To
a large degree, this is because it has somewhat been neglected to study the formal semantics of the involved planning languages. In fact, there was not any research on a formal account of STRIPS until the mid-1980s [Lif87]. Moreover, the semantic definitions that were proposed heavily rely on meta-theoretic operations on logical theories. It was not until the 1990s that the first purely declarative semantics for STRIPS was presented by Lin and Reiter [LR97] who show the correspondence of STRIPS operators to a certain form of Situation Calculus axioms. This does not only provide a deeper understanding of STRIPS that for instance allows it to be generalized to knowledge bases with only incomplete information. It also allows for a semantically well-founded embedding of STRIPS planning into a GoLog system, where the latter is responsible for the overall control, including sensing updates and plan monitoring. It seems desirable to be able to extend this approach to more expressive sublanguages of PDDL in order to exploit the most recent progress in state-of-the-art planning systems.

1.1.2 Verification

Before deploying a GoLog program to the actual robot and executing it in the physical world, it is often desirable if not crucial to verify that it indeed fulfills its intended purpose and meets certain requirements such as safety, liveness and fairness conditions. For example, we may want to ensure that every request for coffee is eventually served by the robot, and that it never runs out of energy, but always returns to its charging station in time. In this respect, the design and implementation of a GoLog agent is no different from industrial hardware and software development in general, where verification is an important and critical step in the overall development process.

In many scenarios, in particular that of an autonomous robot performing an open-ended task, the corresponding control program is a non-terminating one. Surprisingly, the verification of non-terminating GoLog programs has so far received very little attention within the Situation Calculus research community, with a few notable exceptions. De Giacomo, Ternovska and Reiter [GTR97] show how the semantics of non-terminating processes can be defined by means of (second-order) Situation Calculus axioms, how properties can be expressed using second-order fixpoint formulas, and that it is then possible to prove the satisfaction of these properties given the aforementioned axioms.

Apart from the fact that second-order formulas, inductive fixpoint definitions and the underlying μ-calculus are typically difficult to grasp even for the mathematically inclined, and all the more so for a GoLog programmer who may not be an expert in logic, problematic about this approach is that properties are proven manually, which is not only tedious, but also prone to errors. It would rather be desirable if verification could be done automatically by means of
appropriate algorithms.

The formal verification of non-terminating processes has been the subject of research on model checking [CGP99, BK08] since its introduction in the mid-1980s. Typically, the system is represented through some kind of graph structure (its model) and the property in question is expressed in some temporal logic. In addition to the fact that this kind of representation is very human-comprehensible, it allows to basically reduce the satisfaction of the property to a graph-theoretical problem, and thus solve it efficiently.

In the face of the abundance of formalisms and methods the field has come up with, it is conceivable to use existing model checking tools for the verification of non-terminating GOLOG programs. However, the underlying formalisms are usually chosen very carefully to ensure decidability or even tractability of the method. Consequently, they are of very restricted expressiveness, in particular regarding first-order quantification, which is either supported only in a very limited fashion, or not at all. On the other hand, as argued above, the first-order expressiveness of the Situation Calculus is considered a desirable feature that one would rather not give up, but the reason why we chose the language in the first place. It seems preferable to be able to do the verification in the very same expressive formalism and with the very same reasoning methods that we use for the actual control and specification of our agent.

1.2 Objectives and Contributions

For the above stated reasons, the goal of this thesis is to study both planning and verification in the agent language GOLOG. In both cases, this is to be done in a semantically well-founded, preferably declarative manner, as opposed to a meta-theoretic ad-hoc integration. In particular, the contributions of this thesis are as follows:

- **Foundations for knowledge-based agents using $\mathcal{ES}$**
  A formal account of knowledge-based agents on the basis of the logic $\mathcal{ES}$ is provided. This includes the integration of various reasoning procedures to handle actions, sensing and knowledge, an $\mathcal{ES}$-based transition semantics for a rich variant of GOLOG, as well as a formal interface definition for the meta-level control that implements the sense-plan-act cycle of the agent.

- **Declarative semantics of PDDL in $\mathcal{ES}$**
  The semantics that has been provided for the planning language PDDL so far [FL03] is only meta-theoretic and at times hard to grasp. Similar to Lin and Reiter’s treatment of STRIPS, in this thesis, a declarative semantics for an expressive subset of PDDL
is presented that not only helps to further our understanding of the language, but that is also the basis for the embedding of efficient state-of-the-art planners into GOLOG. An empirical evaluation shows that such an embedding is indeed highly advantageous in terms of the increase of the overall runtime performance of the system.

- **The logic \( \mathcal{ES} \)**
  An extension of the logic \( \mathcal{ES} \) is given that includes new modal operators in order to express properties of both terminating and non-terminating GOLOG programs. With a syntax resembling classical dynamic logic [HKT00], the new modalities contain GOLOG programs as arguments whose execution traces they quantify over. To describe or constrain these traces, formulas may moreover contain modal operators known from temporal logics [Eme90] such as “always”, “eventually”, and “until”. Similar in spirit to the original \( \mathcal{ES} \), the meaning of these operators is defined within the logic’s semantics, rather than axiomatically.

- **Verification algorithms for \( \mathcal{ES} \)**
  Finally, algorithms are provided for the automated verification of terminating and non-terminating GOLOG programs. The presented procedures constitute reasoning methods for different subsets of \( \mathcal{ES} \) formulas. They rely on a newly introduced graph representation for GOLOG programs called characteristic graphs in order to systematically explore the state space by means of an iterative fixpoint approximation. Similar to other forms of reasoning in the Situation Calculus, the verification of \( \mathcal{ES} \) formulas thus ultimately reduces to classical first-order theorem proving. Because of the semi-decidability of FOL, only the soundness of the methods can be guaranteed.

### 1.3 Outline

The remainder of this thesis is organized as follows:

**Chapter 2** presents related work for the areas of action logics, planning and verification. Its purpose is to provide a rough overview of the fields, without making any claim to be exhaustive. Work more specifically related to the contents of this thesis will be discussed within the individual chapters.

**Chapter 3** gives the logical foundations for the remainder of this thesis. After motivating its use, the syntax and the semantics of the logic \( \mathcal{ES} \) is introduced, followed by the \( \mathcal{ES} \) variants of established Situation Calculus concepts such as basic action theories, regression and
progression. Furthermore, reasoning with belief modalities and sensing is discussed, after which the language Golog is presented, alongside with an $\mathcal{E}$-based transition semantics. Finally, it is shown how all these ingredients can be put together to build a knowledge-based agent. Parts of this chapter are based on [CL06a].

Chapter 4 is dedicated to planning in Golog. It starts with a formal definition of a declarative semantics in terms of a mapping to $\mathcal{E}$ basic action theories for an important fragment of the planning language PDDL that extends basic STRIPS by arbitrary preconditions and conditional effects, among other things. Afterwards, the thus theoretically justified embedding of state-of-the-art PDDL planners in Golog is discussed and evaluated empirically. This chapter is based on material that was previously published in [CL06b], [ENLC06], [CELN07], [CELR08], and [CRLN12].

Chapter 5 deals with the verification of Golog programs, beginning with the language definition of the logic $\mathcal{E}\mathcal{S}$, followed by the introduction of the concept of characteristic graphs as a means for representing Golog programs. The main results are then provided in the form of verification algorithms for different subsets of $\mathcal{E}\mathcal{S}$. First, properties of non-terminating Golog programs are considered that are expressed in a fragment that resembles a first-order variant of the classical temporal logic CTL. Next, a more expressive class of properties is addressed that is similar to CTL$^\ast$. Finally, we discuss the verification of postconditions of terminating Golog programs and sketch an implementation based on binary decision diagrams. This chapter is an extended version of what was presented in [CL08] and [CL10].

Chapter 6 concludes with a summary and an outlook on future work.
Chapter 2

Related Work

The purpose of this chapter is to put this thesis and its contribution into the context of past and current research on action formalisms (Section 2.1), planning (Section 2.2), and agent verification (Section 2.3).

2.1 Action Formalisms

The idea of using a logical formalism for building intelligent agents is almost as old as the field of Artificial Intelligence itself. McCarthy [McC59] envisioned a system called advice taker that represents information about the state of the world, the system’s capabilities as well as its goals declaratively, i.e. by means of propositions expressed as formulas in some logical language. As opposed to classical imperatively programmed systems in which all eventualities need to be foreseen by the programmer, advice taker would be able to deduce the right course of action by inferring it from the formulas in its memory.

The advice taker however was not much more than an idea, and McCarthy’s description of its underlying language and deduction method was rather sketchy. Subsequent work on defining an appropriate formalism eventually led to the introduction of the Situation Calculus in [McC63, MH69], where McCarthy and Hayes discuss how a dialect of first- (and higher-) order predicate logic can be used to express facts about the state of the world, the abilities of an agent, and the effects of its actions.

Although their presentation was again still at an informal level, they already recognized a major challenge in the logical formalization of an agent, which they called the frame problem: To be able to reason about whether a certain course of action will bring about some goal, it is not sufficient to only encode what changes result as effects of an action, but also what remains unchanged by it. While an encoding of the direct, intended effects of an action is typically
straightforward as there are usually only few of them, manually identifying and formalizing all invariants is a somewhat hopeless endeavour because of their sheer number. As an example, the action \texttt{open(window)} will cause the \textit{window} to be opened, but it will leave unaffected what the colour of the window is, whether the door is open or closed, where the furniture is positioned, what the weather is like, who the chancellor of Germany is and so on.

The frame problem was not the only hurdle the Situation Calculus had to overcome before achieving its today’s prominence. In his seminal paper, Green [Gre69] proposes to represent a mobile robot’s actions and environment by Situation Calculus axioms and then generate plans using constructive resolution-based theorem proving. The pioneering robotic project \textit{Shakey} at SRI (\textit{Stanford Research Institute}) was first based on Green’s account of planning. Due to the approach’s computational inefficiency it was however quite soon abandoned in favor of the STRIPS (\textit{“Stanford Research Institute Problem Solver”}) system [FN71], which allowed for a more efficient search-based planning method at the cost of limited expressiveness: action effects were simply represented by lists of literals that have to be added to or deleted from the description of the current world state. When people nowadays speak of STRIPS, it is usually this representation language that they are referring to, rather than the underlying algorithmic approach, as the former has been far more influential than the latter. Nonetheless it is justified to say that the work on STRIPS gave rise to what has become today one of the most important subareas of AI, namely that of \textit{planning}, which we will discuss in more detail in Section 2.2.

Since then research on action formalisms developed rather independently from that on planning, despite the two fields’ common origin. Whereas work on planning was more focused on developing efficient and practicable systems, action logic researchers concentrated more on expressive formalisms and the fundamental theoretical issues underlying them. In particular, the Situation Calculus gained new momentum with a solution to the above mentioned frame problem that is due to Reiter [Rei91], who combined ideas from earlier proposals by Pednault [Ped89], Davis [Dav90], Haas [Haa87], and Schubert [Sch90]. Alongside with the so-called \textit{successor state axioms} that constitute the core of Reiter’s solution, he presents a corresponding \textit{regression} method that allows to transform a formula about what will be true after doing certain actions into an equivalent formula only describing the current situation. Since regression is a purely syntactic manipulation, reasoning about actions and change thus becomes easier as all dynamic aspects can be eliminated before the actual theorem proving is applied.

Regression is thus a solution to the \textit{projection problem}, which means to decide whether some formula will hold after executing a sequence of actions, given a background axiomatization of the current situation as well as preconditions and effects of actions. While regression solves this problem quite elegantly, it also has its drawbacks. First, the size of the regression result
usually grows exponentially with the number of actions involved. Regression thus soon becomes infeasible as a robot that has been operating for some time has typically accumulated a rather long history of actions. Furthermore, such a history likely contains many redundancies, for example an \texttt{open(window)} action that is later on undone by means of \texttt{close(window)}. An alternative to transforming a query property backwards is to instead update the knowledge base forwards through the actions that have been performed so far, which is commonly referred to as \textit{progression}. The latter lets an agent keep only information that is relevant for the present and future and “forget” everything else. Ideally, a robot performs progression during phases of “mental idle time” where it is busy performing physical actions (like moving from one room to the next), but is not occupied cognitively. Based on Reiter’s formalization of successor state axioms, Lin and Reiter [LR97] present a model-theoretic account of progression and study its properties. Unfortunately, their analysis comes with a strong negative result: In general, a first-order knowledge base may not possess a progression that is in turn expressable through first-order logic. However, they discuss a variety of restricted classes of theories for which a first-order progression is guaranteed to exist, including one that corresponds to the above mentioned STRIPS formalism.

Today, the formalization of the Situation Calculus developed by Reiter and his colleagues at the Toronto Cognitive Robotics Group [LPR98, PR99, Rei01a] is the most widely used and deeply studied one. Among other things, it has since been extended to include a variety of features such as indirect effects [Lin95, Lin96], continuous processes [Pin94, Rei96, Pin98], and a probabilistic notion of uncertainty [BHL95]. To account for the fact that real-word agents only possess incomplete information and have to be able to reason about their knowledge and how it is affected by their actions, Moore [Moo79, Moo85] adapted the standard possible-world model of knowledge from epistemic modal logic [Hin62, Kri63] for the Situation Calculus. Scherl and Levesque [SL93, SL03] presented a solution to the frame problem for knowledge, alongside with a corresponding regression method. Lakemeyer and Levesque propose the logic \texttt{ES} [LL04, LL05a] as an alternative that overcomes some shortcomings of earlier formalizations of knowledge and action in the Situation Calculus by using modal operators for action and knowledge similar to dynamic logic [HKT00] and epistemic logic [Kri63], respectively, and by fixing their meaning semantically rather than axiomatically.

Reiter’s concise solution to the frame problem also lead to the development of the high-level agent programming language \texttt{Golog} (for \texttt{aLGOL in LOGic}) [LRL+97]. It allows to construct complex behaviours out of high-level actions by combining them on the one hand through control structures known from imperative programming languages such as conditionals, loops, and recursive procedures. On the other hand, \texttt{Golog} programs can also contain nondeterministic
constructs that leave choices to the system and have to be resolved through some form of lookahead. Hence, the programmer can freely combine programming and planning within a GOLOG program such that the system concentrates its computational effort where it is actually needed. Semantically, GOLOG is defined by a meta-theoretic operation that macro-expands a program to a Situation Calculus formula. In principle, all it thus takes to build a GOLOG interpreter is an implementation of Reiter’s regression mechanism together with a theorem prover.

Since its first introduction, numerous variants of GOLOG have been proposed. ConGOLOG [GLL00] extends the original language by concurrency, interrupts, and exogenous actions. It also comes with an alternative semantics where second-order Situation Calculus axioms define single-step transitions of programs and the conditions under which they can terminate. Lake-meyer’s sGOLOG [Lak99] is the first approach to include sensing into GOLOG, where instead of producing a linear sequence of actions, the interpreter yields a tree of actions that branches over possible outcomes of sensing actions. Reiter’s sequential temporal GOLOG [Rei98] incorporates an explicit notion of time. Finzi and Pirri [FP04] study a concurrent variant which uses constraint-based interval planning. IndiGOLOG [GL99, GLS01, GLLS04] is especially suited for realistic scenarios as it overcomes some of the drawbacks of earlier variants. First, the original GOLOG executes programs offline, meaning that before actions are executed physically, the interpreter searches for a sequential solution for the entire input program. As this may be prohibitively time consuming, in particular for larger, non-trivial programs, IndiGOLOG resorts to an online approach that interleaves lookahead with action execution. Moreover, the language allows the agent to use sensing actions to gather new information at runtime. A variant for knowledge-based programs [Rei01b] uses Scherl and Levesque’s solution to the frame problem for knowledge. ccGOLOG [Gro02, GL03] introduces continuous change based on the temporal Situation Calculus. DTGOLOG [BRST00, Sou01] is a decision-theoretic variant where actions can have probabilistic outcomes and the system searches for an optimal policy that maximizes the expected reward for the agent. The game-theoretic GTGOLOG [FL04] generalizes DTGOLOG to adversarial multi-agent scenarios. READYLOG [Fer07, FL08] aims at highly dynamic real-time domains such as robotic soccer.

There are many other action formalisms aside from the Situation Calculus. Thielser’s Fluent Calculus [Thi99, ST00] is derived from the Situation Calculus, but solves the frame problem in a different manner. As opposed to using one successor state axiom per predicate that is affected by actions, his solution relies on having one state update axiom per action that describes how a state is changed by that action. FLUX [Thi05] is a run-time system for the Fluent Calculus that plays a similar role as GOLOG does for the Situation Calculus. Schiffel and
Thielscher [ST05, ST06] explore the formal connections between Situation Calculus and GOLOG on the one hand and Fluent Calculus and FLUX on the other hand. One successful application of FLUX is in General Game Playing [ST07], where the task is to develop computer programs that are able to play any previously unknown game without any human intervention, simply by providing them with a formal description of its rules.

Furthermore, the Event Calculus [KS86, Sha95, Mue06a] is a narrative-based action formalism utilizing an explicit notion of time, where planning is done in the form of an abductive reasoning task that relies on Circumscription [McC80]. Temporal Action Logic (TAL) [DGKK98, DK01b] is derived from Sandewall’s Features and Fluents [San94] framework and represents time in a similar quantitative manner, but uses a non-classical surface language to represent narratives and translates them to a classical base language in order to perform Circumscription-based reasoning. The similarities and differences between the Event Calculus and TAL are explored in [Mue06b]. APL [HdBvdHM99] is an agent programming language based on the beliefs-desire-intention (BDI) model [Bra87] which combines features from imperative and logic programming, using a transition-style semantics. The action logic $A$ [GL93] and its successors $B$ and $C$ [GL98] as well as $C+$ [GLL+04] use causal laws to describe dynamic domains, where plans are generated by means of Answer Set Programming (ASP) [Gel08]. Dynamic Logic [HKT00] was originally intended for reasoning about computer programs, but has also become a popular means for describing agents, in particular its propositional variant [FL79]. Epistemic Dynamic Logic shares many similarities with the Situation Calculus and ES [vDHL11].

2.2 Planning

In AI, planning (in its classical sense) refers to the deliberation process of finding a sequence of actions that achieve a desired goal, given a description of the current state of the world and the action operators at the agent’s disposal. Planning has been a central research topic ever since the earliest years of AI. The above mentioned STRIPS [FN71] is widely viewed as the first major planning system. Its underlying algorithm is based on the General Problem Solver GPS [NS63], a state-space search system using means-end analysis. However, the STRIPS language has been far more influential than its algorithm, even up to the present day, as the majority of planners that were since developed were based on some variant, extension or derivative of it.

The success of STRIPS was in large part due to its simplicity. Action operators are described in terms of three components: preconditions, positive effects, and negative effects, each of which is given in the form of a list of logical atoms. Thus, it is straightforward to determine
the result of an action to a state given in the form of a set of atoms, namely by adding all atoms that are positive effects (hence they are also called the “add list”) and deleting all those that are negative effects (“delete list”). While this made planning possible in the first place, expressiveness was very restricted, and early systems such as the first linear planners of the 1970s [Sus75] could only handle rather short plans [Byl94]. The following two decades were mainly dominated by partial-order planners [PW92], until a major breakthrough was made in the mid 1990s with the development of the GraphPlan method [BF97], which yielded a performance increase of several orders of magnitude.

An early proposal for a formalism that goes beyond the limited expressiveness of STRIPS was the Action Description Language ADL [Ped89, Ped94] which extends basic STRIPS by features such as conditional effects, functions and quantification, among others. One could argue that most planners that were developed at the time supported either an extension of STRIPS or a subset of ADL as input language. UCPOP [PW92] for example could handle problems that contain universal quantification and conditional effects (hence the letters ‘U’ and ‘C’ in the acronym), but neither existential quantification nor disjunction. Since each planner used its own syntax, only few were directly evaluated against each other, and it was therefore difficult to judge the overall progress in the field.

This changed with the introduction of the biennial International Planning Competition (IPC) [McD00], first held at the 1998 AI Planning Systems conference (AIPS), which aims at creating a repository of planning domains and problems, to provide a benchmark for planners that allow researchers to compare their system to the state of the art, and to focus and drive research towards more realistic applications. The Planning Domain Definition Language PDDL [GHK+98] that is used as the common input language at the competitions has by now become the de-facto standard for the formulation of planning domains and problems. It extends basic STRIPS by features of ADL. Moreover, later extension include numerics, durative actions and concurrency [FL03], derived predicates and timed initial literals [EH04], as well as quantitative preferences and trajectory constraints [GL05b]. Since most of these features are tackled in special competition tracks, planning systems do not have to support all of them but can specialize on certain fragments. Furthermore, apart from the most prominent case of classical planning under complete information, there are now also tracks for learning systems and planners that can deal with probabilistic uncertainty.

Since the introduction of the IPC and PDDL, one could observe a sudden thrust of progress in the area. In the case of classical planning, today’s state-of-the-art planners apply a multitude of different paradigms. One popular approach is to convert the planning instance into a propositional satisfiability problem and apply solving techniques that have been developed in
this area [KS92, KS96]. A related idea is to use methods for constraint satisfaction problems
[DK01a]. Many systems however apply a forward search algorithm in the state space in combi-
nation with a sophisticated heuristics [BG01b, BG01a] that is in most cases based on the
groundbreaking GraphPlan algorithm [BF97]. Two of the fastest such planners are FF [HN01]
and Fast Downward [Hel06a] which form the basis of many current successful planning systems.
Instead of searching in the state space, some systems do a search in the space of partial plans
[PW92, GS02].

The practical progress that has been made in developing efficient planners has been comple-
mented by theoretical studies on planning in terms of the expressiveness of planning formalisms
and the computational complexity of the planning problem. In the worst case, determining
whether a planning problem has a solution is very hard. Bylander [Byl94] showed that proposi-
tional STRIPS planning is PSPACE-complete. Erol, Nau and Subrahmanian [ENS92, ENS95]
furthermore study how decidability and computational complexity of regular STRIPS is af-
fected by restrictions on the language such as disallowing function symbols, delete lists, or
negated preconditions. In practice, worst case complexity however often does not apply due to
the fact that many instances of planning problems are in fact easier. Studies on the complexity
of various planning domains were done by Gupta and Nau [GN91] and Helmert [Hel03, Hel06b].
Moreover, the computational complexity of deciding plan existence may be too coarse a mea-
sure when it comes to comparing the expressiveness of different formalisms, as the length of
plans may also play a role. Bäckström [Bäc95] proposes to use structure-preserving many-one
reductions for this purpose, however his assumptions that plan lengths remain unchanged
and that reductions are computable in polynomial time are sometimes too strict. Nebel’s [Neb00]
compilation schemes circumvent this problem by allowing plans to also grow linearly or poly-
nomially, and by only requiring the the compiled instances to be of representable in polynomial
size. He demonstrates the usefulness of these compilation schemes by presenting an exhaustive
study of the expressiveness and compilability of different subsets of propositional ADL.

When some or all of the restricting assumptions of classical planning are dropped, one
enters the area of planning under uncertainty. When actions have nondeterministic outcomes,
but states are fully observable, conditional planning [PS92] has to be applied, where the aim
is to find a policy (a mapping from states to actions) that achieves the goal, and, in case that
outcomes are assigned probabilities, that also minimizes plan costs. In the case of conformant
planning, the task is to find a sequence of actions that achieves the goal from any initial state
and under any outcome of a nondeterministic action, given that states are not observable at all
[STGM05]. Moreover, domains where action outcomes are stochastic are typically formalized
and solved in terms of Markov Decision Processes (MDPs) [Bel57, Put94, BDG00] or Partially
Observable MDPs (POMDPs) [Mon82, BP96]. Furthermore, quantitative or qualitative user preferences may have to be considered [BM08]. Recently, there has also been growing interest in generalized planning which aims at solving entire problem classes by synthesizing controllers that may even contain loops [Lev05, HG11].

All systems and approaches mentioned so far (which by no means constitute an exhaustive list\(^1\)) can be subsumed under the term of domain-independent planning, which means that the corresponding planners are not specialized or fine-tuned to any particular planning domain. In many application scenarios it is however often the case that the human user or designer possesses special knowledge that might be helpful to the planner in restricting the search space, thus yielding an increase in efficiency. Domain-dependent planners therefore allow to provide such domain knowledge in addition to the actual planning instance that is to be solved. TLPlan [BK00] is able to make use of control information that is given in the form of formulas of linear temporal logic (LTL), whereas TALplanner [DK01b] is based on the above mentioned TAL. Planners using Hierarchical Task Networks (HTN) [EHN94] such as SHOP2 [NAI+03] also fall into the domain-dependent category as well; the underlying idea is to provide the system with knowledge of how to decompose tasks into subtasks, which might consist of further subtasks or atomic actions. Furthermore, behaviours and strategies that are formulated in agent programming languages such as GOLOG constrain the search space and thus also constitute a form of domain-dependent control knowledge.

However, sometimes such information may not be available. Although GOLOG supports pure, domain-independent planning in principle, existing GOLOG systems can typically not compete performance-wise with current PDDL planners. The reason lies in the fact that the former usually resolves non-determinism by blind search, whereas the latter resort to a variety of sophisticated techniques and heuristics, as indicated above. In face of the abundance of efficient PDDL-based planning systems, it suggests itself to try to exploit these achievements and apply them in the context of GOLOG, with the aim of thus acquiring a system which is both expressive and able to efficiently solve planning problems.

As a prerequisite to this endeavour, we have to be able to relate the two formalisms in terms of their semantics. As opposed to the field of action logics, semantics played only a minor role in the area of planning. For a long time, there has not been any formal semantics for STRIPS at all, until the first meta-theoretic formalizations were presented [Lil87, ENS92, BY94] that are based on the transformation of logical theories. Later, embeddings into the Situation Calculus were proposed [Ped89, SS98], which are much more appealing as these represent declarative

---

\(^1\)Past and current IPC results as well as information on the participating planning systems can be found at [http://www.icaps-conference.org.](http://www.icaps-conference.org)
2.3 Verification

definitions in terms of a well-defined logic whose properties are well understood. In particular, Lin and Reiter [LR97] relate the state updates of STRIPS to the progression of certain Situation Calculus theories, thus combining the benefits of a declarative definition with the efficiency of STRIPS operators. Moreover, their approach also works in the more general case of incomplete theories.

It seems natural to try to generalize these results to more expressive languages such as PDDL, which was originally not accompanied by a formal semantics, but rather intended as a standard syntax for a commonly accepted semantics for STRIPS, as described by Lifschitz [Lif87]. The first formal semantics was provided for version 2.1 by Fox and Long [FL03], who extended Lifschitz’ state-transitional definition to also cope with the newly added numeric and temporal features of PDDL. In any case, the meaning of action operators is defined only meta-theoretically through the addition and deletion of literals, which already Lin and Reiter identified as problematic in the case of STRIPS, in particular for logically incomplete theories. Moreover, the formalization is rather complicated and therefore quite difficult to grasp for a human reader. On the other hand, a declarative semantics for PDDL based on the Situation Calculus or ES would not only further our understanding of the language, but also provide the basis for the combination and integration of PDDL planners with Golog. Chapter 4 of this thesis will address this issue.

2.3 Verification

In computer science, formal verification refers to the act of proving or disproving the correctness of a system with respect to a formal specification. Verification is a critical and major part of the overall process of industrial hardware and software development. Before deploying a product, it has to be ensured that it indeed fulfills its intended purpose and meets certain requirements. Not only does it become more costly to fix an error the later in the development process it is detected, but hardware and software reliability is also often directly related to the safety of human beings. The design and implementation of a knowledge-based robot is no different in all of the above respects.

First ideas on verifying the correctness of (terminating) programs were already presented by Turing [Tur49, MJ84], foreshadowing what would later be called the axiomatic approach on program verification as formalized by Floyd [Flo67] and Hoare [Hoa69]. The approach requires the programmer to annotate the program by assertions (pre- and postconditions, loop invariants) in the form of logical formulas, and a given proof system consisting of axioms and inference rules then allows to prove program properties formally, typically by means of an interactive,
higher-order logic theorem prover. Examples where this approach has been put into practice include the correctness proof of a large fragment of the programming language C [Ler09] using the theorem prover Coq [BC04], as well as a correctness proof of the seL4 micro kernel [KEH+10] by means of the Isabelle/HOL prover [PW02]. Other theorem provers such as KeY [BHS07] and KIV [BRS+00] formalize Dynamic Logic [HKT00], which can be viewed as a generalization of classical Hoare Logic.

When it comes to the formal verification of non-terminating processes, the most prominent current approach is probably model checking [CGP99, BK08], which was introduced in the early 1980s [CE81, CES86, QS82]. In the classical case, the system is represented by a model that is given in the form of a finite graph structure, and the properties to be verified are expressed in some temporal logic such as CTL, LTL, or CTL∗ [Eme90]. It is then possible to check the formula in question by a systematic, graph-theoretic analysis of the model. A major performance boost to model checking methods was achieved in the mid-1990s with symbolic model checking [BCM+92, McM93]. By resorting to compact, implicit representations such as ordered binary decision diagrams (OBDD) [Bry86], it is possible to handle very large state spaces. With appropriate finite representations, even infinite-state systems can be handled, for instance using pushdown automata [BEM97], petri nets [Esp94], or variants of Minsky’s counter machines [Dem06]. Moreover, while originally model checking typically used propositional formalisms to ensure decidability, approaches have been proposed for the verification of first-order specifications, for example by sufficiently restricting expressiveness to ensure decidability [WTM04] or by combining symbolic model checking with theorem proving and taking into account that the method is not guaranteed to terminate [BDG+98]. Furthermore, infinite state spaces can be reduced to finite ones by means of predicate abstraction [CDG01, BP06], and can then be verified using any common finite model checking tool such as NuSMV [CCG+02] or SPIN [Hol03].

Formal verification lends itself particularly well to knowledge-based agents because the system is already defined in a formal, logical manner. Ideally, it is hence unnecessary to come up with a system description manually, which is prone to errors. It is rather possible to do the verification within the very same logical representation and with the same reasoning tools that are used for the actual control of the agent, as was already pointed out by Green [Gre69].

Regarding the formal verification of GOLOG programs, one observation is that if the agent performs an open-ended task as in our envisioned domestic robot scenario, it will (at least ideally) operate indefinitely. Its corresponding control program therefore has to be a non-terminating one. Surprisingly, the verification non-terminating GOLOG programs has so far received very little attention within the Situation Calculus research community. A notable exception is a paper by De Giacomo, Ternovska and Reiter [GTR97], who define programs and
properties in terms of second-order formulas and then do a manual, meta-theoretic proof to show that a program indeed satisfies a given property. This is problematic for at least two reasons. On the one hand, their formulation in terms of second-order fixpoint formulas is quite involved and difficult to grasp even for the mathematically inclined. On the other hand, manual proofs are of course tedious and prone to errors, and it would be much more desirable to be able to do an automated verification, using an appropriate algorithm. In the face of the abundance of results on model checking and available tools, it is conceivable to utilize them for the verification of GOLOG programs. However, the formalisms they use are often of very restricted expressiveness and typically tailored to low-level computer programs rather than high-level agent programs. Instead of a cumbersome translation from GOLOG to the model checker’s input language, which may result in a loss of expressiveness, it seems preferable to be able to follow the above mentioned idea of verifying GOLOG directly within its underlying logic. Chapter 5 of this thesis will be concerned with this subject.

Although there has not been much work on the verification of non-terminating GOLOG programs so far, related forms of reasoning in the Situation Calculus have been studied. Liu [Liu02] presents a proof system in the style of Hoare logic for proving properties of terminating GOLOG programs. Shapiro, Lespérance and Levesque [SLL02] describe CASLve, a framework for verifying properties of multi-agent systems that are specified in CONGOLOG, where an axiomatization is fed into a theorem prover in order to do interactive, computer-aided proofs. Gu and Kiringa [GK06] represent classical propositional model checking within the Situation Calculus. Reiter [Rei93] proves properties that hold throughout all world states by means of induction. Bertossi et al. [BPS+96] propose an automatic constraint verification system using an induction theorem prover. Kelly and Pearce [KP07, KP10] provide an algorithm for checking the persistence of formulas. An alternative approach to checking and proving state constraints is to ensure them by compiling them into the agent’s axiomatization [LR94b, McI00]. If a system shows unexpected behaviour, *diagnosis* [McI98, Iwa02, SBM10] can further be used to generate possible explanations that help to identify possible sources of failure. Finally, verification has been studied in other agent languages as well. To mention but a few, de Boer et al. [dBHvdHM07] provide a verification framework for the propositional agent programming language GOAL. Van der Hoek, Lomuscio and Wooldridge [vdHLW06] study the complexity of model checking properties formulated in the Alternating-Time Temporal Logic ATL [AHK02] for models expressed in the Simple Reactive Modules Language SRML. Alechina et al. [ADLM07, ADLM08] discuss the verification of agents formalized in SimpleAPL, a fragment of 3APL that allows the implementation of agents with beliefs, goals, actions, plans, and planning rules.
Chapter 3

Logical Foundations

The purpose of this chapter is to lay the formal foundations for this thesis. Section 3.1 first gives an informal introduction as to why we resort to the logic $\mathcal{E}S$, a modal variant of the classical Situation Calculus. The formal definition of $\mathcal{E}S$ is then given in Section 3.2. Section 3.3 presents how the logic can be used to axiomatize dynamic domains by means of so-called basic action theories. Regression and progression, which are two different forms of reasoning with dynamic properties, are discussed in Section 3.4. Section 3.5 presents how epistemic operators are used in $\mathcal{E}S$ in order to express facts about the agent’s knowledge, its lack thereof, and how it is affected by sensing. The agent programming language Golog, which is defined on top of $\mathcal{E}S$, is introduced in Section 3.6. Section 3.7 finally shows how all of the above can be integrated to build a knowledge-based agent system.

3.1 A Modal First-Order Logic for Dynamic Domains

In the previous chapters we argued that a logical representation of an agent has many benefits. Once we have decided to follow the knowledge-based approach, the first question that arises is the choice of an appropriate representation formalism.

3.1.1 First-Order Representation and Reasoning

Recall that an autonomous agent’s knowledge about the state of its environment is typically incomplete. While it may know some facts to be true, at the same time there might be other things about which the agent is ignorant. As a running example let us consider a mobile robot acting in an office environment. Said robot may be unaware whether the person named Bob is currently in his office or in the kitchen. Using standard logical notation, this can be expressed
by the following formula:

(3.1) \[ \text{In}(\text{bob}, \text{office}(\text{bob})) \lor \text{In}(\text{bob}, \text{kitchen}) \]

While this example statement is still expressible by means of a Boolean combination of ground propositions, this is in general not sufficient. An autonomous agent whose task is open-ended is in most cases not aware of every single individual and object that it potentially may have to deal with in the future. On the contrary, a suitable representation language for the agent’s knowledge base should account for possibly unknown individuals. For example the office robot may know that there is some person other than Bob currently in Bob’s office (and the robot may in fact not know who that person is):

(3.2) \[ \exists x. \text{In}(x, \text{office}(\text{bob})) \land \text{Person}(x) \land (x \neq \text{bob}) \]

We therefore settle for a logic that includes first-order quantification and equality, as this gives us sufficient expressive power to represent knowledge that is potentially incomplete, both regarding the truth of facts as well as the identity of individuals and objects.

One may now be tempted to move from first- to second-order logic, as the latter further increases expressiveness by not only allowing to quantify over single individuals, but also over sets and relations on them. The problem however is the well-known consequence of Gödel’s incompleteness theorem [Göd31] that second-order logic (SOL), as opposed to first-order logic (FOL), does not admit a complete proof theory. This is to say that the validity of every FOL formula (and similarly the logical consequence of a formula from a finite set of premises) is provable, where a proof consists of a finite sequence of statements each of which is either an axiom or follows from some previous statements by means of an inference rule. For SOL this is not the case, i.e. for certain true sentences in the logic there is no, and cannot be any finite proof.

While FOL’s completeness guarantees that a proof of a theorem (a valid statement) can always be found in finite time, such a guarantee cannot be given in the opposite case of an invalid formula. Validity of FOL formulas is hence a semi-decidable problem: Any sound decision algorithm for the validity of FOL formulas will eventually return the answer “yes”, if the input formula is indeed valid, but may not even terminate, if the input is not.

---

1Although this thesis is written with the aspiration to be self-contained in that it gives all relevant formal definitions, we assume that the reader is familiar with the basics of classical predicate logics. A good introduction to mathematical logics can be found in [End72].

2Since semi-decidability also subsumes decidability, we should say more precisely that FOL validity is an undecidable, yet semi-decidable problem.
Semi-decidability implies that no upper bound on the response time of the decision algorithm can be given. Depending on the application domain, this does not necessarily pose a big problem. The purpose of efficient state-of-the-art automated theorem provers such as Vampire [RV02] or E [Sch02] is typically seen in assisting mathematicians in proving mathematical theorems, in which case it is often worthwhile to wait a longer time for the result. However when it comes to the control of a logic-based agent, the possibility that the agent may “get stuck” while deciding on its next action is obviously undesirable. Nevertheless, for the purpose of expressiveness as argued above, we will in the following assume that our agent is one that possesses full first-order reasoning capabilities. However, think of this more as an idealized perspective that we take in order to be able to investigate the involved theoretical issues. In an actual implementation of such an agent one would then need to restrict oneself to some alternative (sub-) logic that is known to be decidable, and thus ensure that the agent will always reach some decision within finite time. One possibility to obtain decidability and even tractability while retaining a significant part of first-order expressiveness is discussed in [CL09].

Despite this assumption of an agent that is a first-order reasoner, below we will equip our logic also with second-order quantification. We will use the latter however only from our “meta-perspective”, i.e. to talk about the agent (e.g. to define concepts, to prove theoretical properties etc.), but do not expect the agent itself to reason with second-order formulas.

### 3.1.2 Standard Names

Now that we opted for a first-order language with equality, there would be good reasons to simply use the “classical” predicate calculus as described in textbooks, most importantly the fact that it has been thoroughly studied for many decades and is therefore well understood. Indeed this approach is taken in Reiter’s Situation Calculus as described in [Rei01a]. Nevertheless there are some drawbacks, one of which arises when reasoning about the identity of individuals.

Consider again statement (3.2) which expresses that there is some unknown person in Bob’s office. The robot now may be told that that person is Bob’s best friend and hence

\[
\text{In}(\text{bestfriend}(\text{bob}), \text{office}(\text{bob}))
\]

holds. But still it is not really justified to say that the agent now knows who the person in Bob’s office is. Indeed the term \(\text{bestfriend}(\text{bob})\) is not sufficient to identify that person, but only gives some further description of it. Assertions involving equalities such as

\[
\text{bestfriend}(\text{bob}) = \text{father}(\text{ann})
\]
do not help either when the agent does not know who the father of Ann is. A similar problem arises even in the case where we use an atomic term such as \textit{mr.smith}, as the identity of \textit{mr.smith} may be still unknown to the robot.

In fact it is a notion of \textit{identity} which regular predicate logics lack. When it comes to “wh-questions” (knowing who, what, when, or where) it is difficult to say whether the agent has an answer since there is nothing that formally distinguishes a “non-identifying” term like \textit{mr.smith} from others such as \textit{bob}. Any further “identification power” we ascribe to the latter is purely informal and meta-theoretical.

For this reason, Levesque and Lakemeyer propose the usage of \textit{standard names} for their logic of only-knowing \textit{OL} [LL01]. The idea is to assume that there is a countably infinite set of names

\[
#1, #2, #3, #4, \ldots
\]

which are syntactically treated like ordinary constants, but which play the role of \textit{unique identifiers}, very much like in database formalisms. If we partition the set of terms into equivalence classes, each standard name corresponds to one such class of all terms equal to it. When the agent then for instance knows that

\[
(3.5) \quad \textit{mr.smith} = #17
\]

we will admit that the agent knows the identity of the individual \textit{mr.smith}, i.e. it indeed knows \textit{who} Mr. Smith is.

Semantically, a standard name such as \#17 does not carry any information whatsoever, except for the fact that it is distinct from any other name such as \#16. Together with epistemic modalities, standard names then allow us to have a formal distinction between \textit{de dicto} and \textit{de re} knowledge, where the former refers to knowing \textit{that} some fact (a formula) is true, whereas the latter means knowing \textit{who} or \textit{what} satisfies a certain property. We will discuss these things formally in Section 3.5. Furthermore, we assume that in every semantical model, the universe of discourse is isomorphic (or indeed identical) to the countably infinite set of standard names. While this yields a non-standard semantics where some of the known properties of standard first-order predicate logic will not hold anymore, it allows us, among other things, to interpret first-order quantification substitutionally. As we will see, the latter not only comes in handy in inductive proofs over the structure of formulas, but also yields a useful property called universal generalization.
3.1 A Modal First-Order Logic for Dynamic Domains

3.1.3 Fluents, Actions and Situations

Predicate logics, be it classical FOL or a non-standard one such as described above, are an appropriate means when it comes to expressing facts and reasoning about objects, their properties, and relations among them. However they take a rather static perspective as there is no explicit notion of possible changes included. Our application domain of an autonomous agent is on the other hand a highly dynamic setting that requires the possibility to encode and model how and when the agent’s actions affect its environment.

For this purpose, our language contains predicates and functions that are assumed to be able to vary as the result of actions. Such “varying” symbols are called fluents, as opposed to non-varying ones, which we call rigid. An example of a fluent predicate (also called relational fluent) is $\text{Holding}(x)$, which we may use to denote that the robot is currently holding object $x$. Obviously this predicate is potentially subject to changes, namely when the robot grabs an object or drops it again. Similarly, an example for a fluent function (also called a functional fluent) is $\text{loc}(x)$, whose value intuitively corresponds to the current location of object $x$, which again is affected when the robot performs moving actions. On the other hand, a predicate such as $\text{NextTo}(x,y)$, denoting that room $x$ is accessible from room $y$, can be considered as rigid since none of the robot’s actions will be capable of altering this property\(^3\).

It is further assumed that indeed all changes to the environment and thus to the fluents’ values are due to the execution of actions, which are represented as terms. Here, function symbols are used as operators, and their arguments correspond to the action’s parameters. For instance $\text{pickup}(x)$ could refer to the action of the robot picking up object $x$ with its robotic arm. We will distinguish actions from other terms by resorting to a sorted logic where each term is assumed to be of one of the sorts action, number, or object, where the latter captures everything that is neither an action nor a number.

Different action formalisms differ in the way in which we may relate actions and fluents within formulas. Reiter’s Situation Calculus makes use of situations, which are basically histories (sequences) of actions. Fluents then have an additional situation argument to distinguish their values in different situations. For example the fact that the robot is not holding the cup initially, but after going to it and picking it up, is expressed as follows:

$$(3.6) \quad \neg\text{Holding}(\text{cup}, S_0) \land \text{Holding}(\text{cup}, \text{do(pickup(cup), do(goto(loc(cup, S_0)), S_0)))})$$

$S_0$ is a constant that denotes the initial situation (before any actions have occurred), whereas

\(^3\)Of course this distinction depends on the frame of reference that we assume. While the adjacency of rooms may be rigid within our office robot running example, it ought to be considered a fluent e.g. in the (far-fetched) case of a construction or demolition robot.
the term
\[ \text{do}(\text{pickup}(\text{cup}), \text{do}(\text{goto}(\text{loc}(\text{cup}, S_0)), S_0)) \]
means the situation resulting from first executing action \( \text{goto}(\text{loc}(\text{cup}, S_0)) \) in \( S_0 \), and then action \( \text{pickup}(\text{cup}) \) afterwards. Note that the nested construction of a situation term using the \textit{do} construct requires us to read the resulting sequence of actions in reverse order.

Apart from expressing facts about specific situations as above, it is also often necessary to describe general laws that hold throughout all situations. For instance we may wish to formalize that whenever the robot performs action \( \text{putdown}(x) \) in situation \( s \), afterwards it will not be holding \( x \) anymore, which can be done by universally quantifying over situations:

\[
\forall s. \neg \text{Holding}(x, \text{do}(\text{putdown}(x), s))
\]

Using situations in this form serves its purpose, but also suffers from some disadvantages, one of which is related to the fact that the Situation Calculus is based on predicate logic with the standard Tarskian semantics, which means that the structure of situations has to be axiomatized. In Reiter’s formalization, so-called foundational axioms are used to ensure that in every model, the situations form an infinite tree of which \( S_0 \) is the root, and where each sequence of actions corresponds to a path to a particular node in that tree, as illustrated in Figure 3.1. Apart from the somewhat cumbersome notation, the problem with this term representation of situations is that model theoretic proofs tend to be quite tedious and involved.

To overcome this problem, Lakemeyer and Levesque proposed in [LL04] to build the tree-like structure of situations directly into the language’s semantics, thus overcoming the need to have explicit axioms defining it. They called the resulting language (for Epistemic Situation Calculus), as it amalgamates the above mentioned epistemic logic \( \mathcal{OL} \) with the Situation Calculus. Unlike an earlier proposal for such a language called \( \mathcal{ACOL} \) [LL98a], \( \mathcal{ES} \) does not contain situation terms at all, but situations remain a purely semantical concept. Instead, two forms of modal operators are used to formulate facts about future situations. On the one hand, \([a] \alpha \) expresses that formula \( \alpha \) will be true after doing the single action \( a \). For instance formula (3.6) would read as follows in \( \mathcal{ES} \):

\[
\neg \text{Holding}(\text{cup}) \land [\text{goto}(\text{loc}(\text{cup}))][\text{pickup}(\text{cup})] \text{Holding}(\text{cup})
\]

On the other hand, \( \Box \alpha \) is used to denote that \( \alpha \) will hold after any sequence of actions. The \( \mathcal{ES} \) analogue of (3.7) thus is:

\[
\Box[\text{putdown}(x)] \neg \text{Holding}(x)
\]
Formally, the restriction to these two modal operators constitutes a loss in expressiveness when compared to the original Situation Calculus. However, as we will see later, the resulting formalism not only yields a simplification for semantic proofs, but it is also still sufficiently expressive to retain the main benefits of the Situation Calculus such as basic action theories, regression-based reasoning and expressing complex actions by means of GOLOG programs.

3.2 The Logic $\mathcal{ES}$

For the above reasons, we decide to employ $\mathcal{ES}$ as the underlying formalism throughout this thesis. It should be noted that since its first proposal in [LL04], different variants and extensions of the language have been introduced. The logic we present below does not exactly coincide with either one in the existing literature, but is a combination of ideas and aspects presented in [LL04], [LL05a], [CL06a], [CHL07], [CL08], [LL09a], and [LL10], with additional adaptations to suit this thesis’ specific needs. We will discuss the details of this at the end of the chapter.
3.2.1 Formal Syntax

Let us start with our logic’s syntax. First of all, the language consists of formulas over symbols from the following vocabulary:

- **object** variables: \( x_1, x_2, \ldots, y_1, y_2, \ldots \);
- **action** variables: \( a_1, a_2, \ldots \);
- **number** variables: \( m_1, m_2, \ldots \);
- **object** standard names: \( o_1, o_2, \ldots \);
- **action** standard names: \( p_1, p_2, \ldots \);
- **number** standard names (constants): \( 0, 1, 2, \ldots, \sqrt{2}, \frac{1}{2}, \ldots \);
- fluent **object** functions of arity \( k \): \( f^k_1, f^k_2, \ldots \);
- rigid **action** functions of arity \( k \): \( g^k_1, g^k_2, \ldots \);
- fluent **number** functions of arity \( k \): \( h^k_1, h^k_2, \ldots \);
- fluent predicates of arity \( k \): \( F^k_1, F^k_2, \ldots \);
- rigid predicates of arity \( k \): \( G^k_1, G^k_2, \ldots \);
- connectives and other symbols: =, \( \land \), \( \land \), \( \forall \), round and square parentheses, period, comma.

That is, we have a countably infinite sets of variables, standard names and function symbols for each of the three sorts. We denote the countable infinite set of **object** standard names by \( \mathcal{N}_O \), the **action** standard names by \( \mathcal{N}_A \), and the **number** standard names by \( \mathcal{N}_N \). The set of all standard names is

\[
\mathcal{N} = \mathcal{N}_O \cup \mathcal{N}_A \cup \mathcal{N}_N.
\]

For simplicity we assume that the functions of sort **object** and **number** are always fluent, whereas the ones of the **action** sort are all rigid. Note that fluent actions are only rarely necessary: Generally we would expect that an action term such as \( \text{pickup}(\text{cup}) \) refers to the very same action, no matter which actions have been performed so far (i.e. which situation we are currently in). In the rare cases where a situation-dependent action is actually needed, for instance when we want to have something like \( \text{bestaction} \), we can simulate a fluent action by using a unary
rigid action symbol \( g(x) \), and then let the argument \( x \) vary accordingly. Similarly, a rigid object or number function can be simulated by using a fluent symbol \( f \) and asserting\(^4\) that

\[
\Box[a](f(\vec{x}) = y) \equiv (f(\vec{x}) = y).
\]

There are both fluent and rigid predicate symbols, and both function and predicate symbols can be of any arity \( k \in \mathbb{N} \) (including zero).

We further assume that the list of number functions above contains the special symbols “+” and “·” that we will use to denote the arithmetical operations of addition and multiplication, respectively. Similarly, the list of rigid predicates is assumed to contain the less-than symbol “<”.

Terms are built as usual, except for the fact that we not only have variables and constants (zero-ary functions) as atomic terms, but also standard names:

**Definition 3.1** (Sorts and Terms). There are three sort: object, action, and number. The set of term, each of which is of one of the three sorts, is the least set such that

- every variable is a term of the corresponding sort;
- every standard name is a term of the corresponding sort;
- if \( t_1, \ldots, t_k \) are terms (of any sorts) and \( f \) is a \( k \)-ary function symbol, then \( f(t_1, \ldots, t_k) \) is a term of the same sort as \( f \).

A term without variables is called a ground term. A primitive term is of the form \( f(n_1, \ldots, n_k) \), where each \( n_i \) is a standard name (of some sort). We denote the sets of primitive terms of sort object, action and number as \( \mathcal{P}_O \), \( \mathcal{P}_A \) and \( \mathcal{P}_N \), respectively. Furthermore \( \mathcal{Z} \) refers to set of all finite sequences of action standard names, including the empty sequence (\( {} \)).

For the arithmetical operations + and · we use the normal infix notation, i.e. we will write \( x + y \) instead of \(+ (x, y)\) and \( x \cdot y \) instead of \( \cdot (x, y)\). The notion of primitive terms is used further below in the semantics to define the denotation of terms, where every term will be mapped to some standard name. We are now ready to define the set of objective \( \mathcal{L}S \) formulas:

**Definition 3.2** (Objective Formulas). The objective formulas are the least set such that

1. If \( t_1, \ldots, t_k \) are terms (of any sorts) and \( P \) is a (fluent or rigid) \( k \)-ary predicate symbol, then \( P(t_1, \ldots, t_k) \) is an objective formula;

\(^4\)This axiom is a special case of a successor state axiom for functional fluents and would have to be included in the basic action theory, see Section 3.3.
2. if \( t_1 \) and \( t_2 \) are terms, then \((t_1 = t_2)\) is an objective formula;

3. if \( \alpha \) and \( \beta \) are objective formulas, \( x \) is a variable, and \( t \) is a term of sort \textit{action}, then \( \alpha \land \beta \), 
   \( \neg \alpha \), \( \forall x \alpha \), \([t] \alpha \), and \( \Box \alpha \), are also objective formulas.

A variable \( x \) is \textit{bound} when it occurs within the scope of a corresponding \( \forall x \) quantifier and \textit{free} when it occurs without being quantified. We call a formula without free variables a \textit{sentence}. If \( x \) is a free variable in \( \alpha \), then \( \alpha^x_t \) denotes the result of simultaneously replacing all free occurrences of \( x \) by the term \( t \). A \textit{primitive formula} is of the form \( F(n_1, \ldots, n_k) \), where each \( n_i \) is a standard name. We denote the set of all primitive formulas by \( P_F \). We use \( \top \) to denote truth, i.e. \( \forall x.(x = x) \), and \( \bot \) to denote falsity, i.e. \( \neg \top \).

The adjective “objective” here refers to the fact that any formula according to the above definition can only express truths about the physical environment, but not the agent’s knowledge about it. The epistemic case is discussed later in Section 3.5. For expressions involving the arithmetical less-than relation, we will again resort to the usual infix notation, i.e. we write \( x < y \) instead of \( <(x,y) \). The greater-than relation \( > \) is to be understood as an abbreviation in terms of less-than:

\[
(x > y) \overset{\text{def}}{=} (y < x)
\]

The reader may also have noticed that our syntax definition did not include all of the logical connectives. As usual, we can define disjunction, implication, equivalence, existential quantification and inequality as abbreviations in terms of other connectives:

\[
(\alpha \lor \beta) \overset{\text{def}}{=} \neg(\neg\alpha \land \neg\beta)
\]

\[
(\alpha \supset \beta) \overset{\text{def}}{=} \neg\alpha \lor \beta
\]

\[
(\alpha \equiv \beta) \overset{\text{def}}{=} (\alpha \supset \beta) \land (\beta \supset \alpha)
\]

\[
(\exists x \alpha) \overset{\text{def}}{=} \neg\forall x \neg\alpha
\]

\[
(t_1 \neq t_2) \overset{\text{def}}{=} \neg(t_1 = t_2)
\]

Thus we view these operators as “syntactic sugar” and do not have to deal with them explicitly in proofs or other definitions.

Furthermore, to reduce the number of parentheses used, we follow the usual conventions of preferences among the logical connectives. Additionally, the \( \Box \) modality has lower syntactic precedence than the connectives, and \([\cdot]\) has the highest priority. In summary, the preference order among the logical operators is as given in Table 3.1. If not stated otherwise, free vari-
Table 3.1: Syntactical preference among logical operators

<table>
<thead>
<tr>
<th>Priority</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>·</td>
<td>¬</td>
<td>∧</td>
<td>∨</td>
<td>∀</td>
<td>∃</td>
<td>⊃</td>
<td>≡</td>
<td>□</td>
</tr>
</tbody>
</table>

The dot notation of quantifiers indicates maximum scope, that is \(\exists x.\alpha \supset \beta\) is read as \(\exists (\alpha \supset \beta)\), whereas \(\exists x\alpha \supset \beta\) stands for \((\exists x\alpha) \supset \beta\).

Two important subtypes of formulas are those that do not contain any actions at all, and those that only contain a limited number of them:

**Definition 3.3** (Static Formulas). A formula \(\phi\) is called static when it contains no \([\cdot]\) and no \(\Box\) operators.

**Definition 3.4** (Bounded Formulas). A formula \(\phi\) is called bounded when it does not contain any \(\Box\) operators.

**Example 3.5.** Consider the following formulas from previous examples:

- (3.16) \(\neg \text{Holding}(\text{cup})\)
- (3.17) \(\neg \text{Holding}(\text{cup}) \land [\text{goto}(\text{loc(\text{cup})})][\text{pickcup(\text{cup})}]\text{Holding(\text{cup})}\)
- (3.18) \(\Box[\text{pickcup}(x)][\text{Holding}(x)]\)

Formula (3.16) is static, and therefore also bounded. (3.17) is a bounded, but not a static formula. (3.18) is neither bounded nor static.

### 3.2.2 Formal Semantics

Now that we established what formulas are, we have to define how to interpret them. That is, we have to devise how to determine the truth value of a formula, given an interpretation. In the case of \(\mathcal{ES}\), an interpretation is given by a *world*: 
Definition 3.6 (Worlds). A world $w$ is a mapping

- $w : \mathcal{P}_O \times Z \to N_O$ and
- $w : \mathcal{P}_A \times Z \to N_A$ and
- $w : \mathcal{P}_N \times Z \to N_N$ and
- $w : \mathcal{P}_F \times Z \to \{0, 1\}$

satisfying the following constraints:

**Rigidity:** If $R$ is a rigid function or predicate symbol, then

$$w[R(n_1, \ldots, n_k), z] = w[R(n_1, \ldots, n_k), z'] \text{ for all } z, z' \in Z.$$

**Unique names for actions:** If $g(n_1, \ldots, n_k)$ and $g'(n'_1, \ldots, n'_l)$ are two distinct primitive action terms, then

$$w[g(n_1, \ldots, n_k), z] \neq w[g'(n'_1, \ldots, n'_l), z] \text{ for all } z \in Z.$$

**Arithmetical correctness:** If $n_1$ and $n_2$ are number standard names, then:

$$w[n_1 + n_2, z] = n_1 + n_2 \text{ and } w[n_1 \cdot n_2, z] = n_1 \cdot n_2 \text{ for all } z \in Z$$

$$w[n_1 < n_2, z] = 1 \text{ iff } n_1 < n_2, \text{ for all } z \in Z$$

Let $W$ denote the set of all worlds.

A world thus maps primitive terms to co-referring standard names of the corresponding sort, and primitive formulas to truth values. The rigidity constraint ensures that rigid symbols do not take different values in different situations, as expected. We further incorporate the unique names assumption for actions into our logic’s semantics, as opposed to the Situation Calculus where this is typically asserted axiomatically. The arithmetical correctness constraint finally means that arithmetical expressions such as $3 + 7$ are mapped to their corresponding value, in this case $10$, and that inequalities such as $5 < 3$ are assigned the expected truth value. Note that the assumption here is that the number standard names coincide with the numerical constants that we may use within arithmetical expressions. We discuss this issue further below in Section 3.2.4.

Definition 3.7 (Denotation of Terms). Given a ground term $t$, a world $w$, and an action sequence $z \in Z$, we define $|t|_w^z$ (read: “the co-referring standard name for $t$ given $w$ and $z$”) by:
1. If $t \in \mathcal{N}$, then $|t|^z_w = t$;

2. if $t = f(t_1, \ldots, t_k)$, then $|t|^z_w = w[f(n_1, \ldots, n_k), z]$, where $n_i = |t_i|^z_w$.

The above definition extends the idea of co-referring standard names from primitive terms to arbitrary ground terms. We now have everything we need to define truth:

**Definition 3.8 (Objective Truth).** Given $w \in W$ and an objective sentence $\alpha$, we define $w \models \alpha$ (read: "$w$ satisfies $\alpha$" or "$\alpha$ is true in $w$") as $w, () \models \alpha$, where for any $z \in \mathcal{Z}$:

1. $w, z \models F(t_1, \ldots, t_k)$ iff $w[F(n_1, \ldots, n_k), z] = 1$, where $n_i = |t_i|^z_w$;

2. $w, z \models (t_1 = t_2)$ iff $n_1$ and $n_2$ are identical, where $n_i = |t_i|^z_w$;

3. $w, z \models \alpha \land \beta$ iff $w, z \models \alpha$ and $w, z \models \beta$;

4. $w, z \models \neg \alpha$ iff $w, z \not\models \alpha$;

5. $w, z \models \forall x.\alpha$ iff $w, z \models \alpha^n$ for all $n \in \mathcal{N}_x$;

6. $w, z \models [t]\alpha$ iff $w, z \cdot p \models \alpha$, where $p = |t|^z_w$;

7. $w, z \models \square \alpha$ iff $w, z \cdot z' \models \alpha$ for all $z' \in \mathcal{Z}$.

The notation $\mathcal{N}_x$ above refers to the set of all standard names of the same sort as $x$. Moreover, $\alpha^n$ means replacing all free occurrences of $x$ by $n$. Reducing the quantificational case to a countably infinite set of instances in this manner is what we call *substitutional quantification*. As we will see later, this is one of the major reasons why inductive proofs in $\mathcal{ES}$ are comparatively simple.

An objective sentence $\alpha$ is called *satisfiable* if some $w$ exists such that $w \models \alpha$. When $\Sigma$ is a set of objective sentences and $\alpha$ is an objective sentence, we write $\Sigma \models \alpha$ (read: "$\Sigma$ logically entails $\alpha$") to mean that for every $w$, if $w \models \alpha'$ for every $\alpha' \in \Sigma$, then also $w \models \alpha$. Finally, we write $\models \alpha$ (read: "$\alpha$ is valid") to mean $\{} \models \alpha$.

### 3.2.3 Properties

Let us now take a look at some of this language’s properties.

**Unique Action Names**

The first interesting property is the uniqueness of action names, as follows:
Proposition 3.9 (Unique Names for Actions). Let \( g \) and \( g' \) be distinct action function symbols. Then

\[
\models g(x_1, \ldots, x_k) \neq g'(y_1, \ldots, y_l) \\
\models g(x_1, \ldots, x_k) = g(y_1, \ldots, y_k) \supset x_1 = y_1 \land \cdots \land x_k = y_k
\]

Proof. This is a direct consequence of the unique names for actions constraint in Definition 3.6.

Note that that the above are identical to the unique names axioms for actions as they are used in the classical Situation Calculus. The difference however is that here they are theorems of our language rather than axioms that we have to assert explicitly.

Comparison with FOL

For better comparing our logic with classical first-order predicate calculus with its standard Tarskian semantics, we restrict ourselves here to a sublanguage without the dynamic aspects of \( \mathcal{ES} \). When we disallow the \([\cdot]\) and \( \Box \) operators as well as any mentioning of actions and numbers (be it as variables, standard names or functions), and when we identify the set of our object standard names \( \{o_1, o_2, o_3, \ldots\} \) with the set \( \{\#1, \#2, \#3, \ldots\} \), we get a sublogic that Levesque and Lakemeyer [LL01] call \( \mathcal{L} \). The first observation is that in terms of semantics, \( \mathcal{ES} \) restricted to \( \mathcal{L} \) formulas coincides with \( \mathcal{L} \):

Lemma 3.10. Let \( \alpha \) be a sentence of \( \mathcal{L} \) and \( \models_{\mathcal{L}} \) mean validity for \( \mathcal{L} \) formulas as defined in [LL01]. Then

\[
\models \alpha \text{ iff } \models_{\mathcal{L}} \alpha.
\]

Proof. (Sketch) Let \( \mathcal{P}_F|_O \) denote the set of all primitive formulas \( F(n_1, \ldots, n_k) \) and \( \mathcal{P}_O|_O \) the set of all primitive object terms \( f(n_1, \ldots, n_k) \), where in both cases all the \( n_i \) are of sort object. A world \( w' \) for \( \mathcal{L} \) according to [LL01] corresponds then to a mapping \( w' : \mathcal{P}_F|_O \rightarrow \{0, 1\} \) and \( w' : \mathcal{P}_O|_O \rightarrow N_O \). Such a world can be identified with an entire equivalence class of \( \mathcal{ES} \) worlds \( w \) that share the same values for \( \mathcal{P}_F|_O \) and \( \mathcal{P}_O|_O \) in the initial situation, but may differ on values for other primitive formulas and terms and in other situations, i.e.

\[
[w'] = \{ w \mid w[F(\bar{n}),()] = w'[F(\bar{n})] \text{ for all } F(\bar{n}) \in \mathcal{P}_F|_O, \\
w[f(\bar{n}),()] = w'[f(\bar{n})] \text{ for all } f(\bar{n}) \in \mathcal{P}_O|_O \}
\]

It is then easy to show that \( w' \models_{\mathcal{L}} \alpha \text{ iff } w \models \alpha \) for all \( w \in [w'] \), the reason being that for the truth of \( \alpha \), only \( \mathcal{P}_F|_O \) and \( \mathcal{P}_O|_O \) matter.
The above result tells us that $\mathcal{L}$ can indeed be viewed as a subset of $\mathcal{ES}$, and we can therefore use any related results, yet stick to our semantics as presented in Section 3.2.2.

One difference between FOL and $\mathcal{ES}$ is in terms of compactness. A logic is compact when a set of formulas is satisfiable if and only if all finite subsets are. It is well known that FOL is compact [Göd29]. However:

**Theorem 3.11** (Levesque and Lakemeyer, [LL01]). $\mathcal{L}$ is not compact.

*Proof.* The infinite set

\[(3.19) \quad \{ \exists x P(x), \neg P(#1), \neg P(#2), \neg P(#3), \ldots \}\]

is not satisfiable, yet any finite subset is. $\square$

**Corollary 3.12.** $\mathcal{ES}$ is not compact.

*Proof.* This is a consequence of Theorem 3.11 and Lemma 3.10. $\square$

This difference is however mostly of theoretical, academic interest, since compactness is a property that is only non-trivial in the context of infinite sets of formulas. In our intended application of devising a logical agent, infinite sets will not play any role as the knowledge base of any implemented agent will always be a finite collection of formulas.

The following theorem shows that $\mathcal{L}$ coincides with FOL when we omit equality and standard names:

**Theorem 3.13** (Levesque and Lakemeyer, [LL01]). Suppose $\alpha \in \mathcal{L}$ does not contain standard names or equality. Then $\models \alpha$ iff $\alpha$ is a valid sentence of ordinary first-order logic.

*Proof. (Sketch)* First note that the proposition of the theorem is equivalent to saying that $\alpha$ is falsifiable in $\mathcal{L}$ iff $\alpha$ is falsifiable in FOL, and that falsifying $\alpha$ is the same as satisfying $\neg \alpha$. The proof of the if-direction now makes use of the fact that for any world $w$ of $\mathcal{L}$ such that $w \models \neg \alpha$, one can construct a corresponding Tarskian model $M$ with a countably infinite universe such that $M \models_{\text{FOL}} \neg \alpha$. In the converse direction, when $M \models_{\text{FOL}} \neg \alpha$, then according to the Löwenheim-Skolem theorem [Sko20], there is a countable $M'$ with $M' \models_{\text{FOL}} \neg \alpha$, for which then a corresponding world $w$ can be constructed with $w \models \neg \alpha$.

In both directions, a countable model $M = \langle D, \Phi \rangle$ corresponding to a world $w$ means that we devise a bijection $\ast$ between $\mathcal{NO}$ and $M$’s domain $D$ such that for all predicate symbols $P$,

$$w[P(n_1, \ldots, n_k)] = 1 \text{ iff } (n_1^\ast, \ldots, n_k^\ast) \in \Phi(P),$$
and for all functions symbols $f$,
\[ w[f(n_1,\ldots,n_k)] = n \iff \Phi(f)(n_1^*,\ldots,n_k^*) = n^*. \]

Things get a little bit more complicated once we add equality into the picture. Consider the infinite sequence of sentences starting with:

1. $\neg\exists x_1\forall y. (y = x_1)$
2. $\neg\exists x_1\exists x_2\forall y. (y = x_1) \lor (y = x_2)$
3. $\neg\exists x_1\exists x_2\exists x_3\forall y. (y = x_1) \lor (y = x_2) \lor (y = x_3)$

The $i$-th sentence in this sequence says that there are more than $i$ individuals. Let $\Delta$ denote the set of all such sentences for $i \in \mathbb{N}$. While FOL indeed allows for models with finite domains, in $\mathcal{E}\mathcal{S}$ and $\mathcal{L}$ we fixed a countably infinite universe of discourse. While this is a somewhat major difference, it does not represent a restriction in terms of applications that are representable. Whenever we want to talk about a finite domain in $\mathcal{E}\mathcal{S}$, it suffices to resort to typed quantification, where some unary predicate $P$ serves as the type:
\[
\forall x (P(x) \supset \alpha)
\]

One then only has to ensure that $P$’s extension remains finite. We will make heavy use of this technique in Chapter 4.

Typically, equality is treated in FOL as a normal predicate for which one has to assert that it is an equivalence relation and which allows the substitution of equals in predicates and functions. More precisely, let $\Gamma$ be the set of the following sentences:

- reflexivity: $\forall x (x = x)$;
- symmetry: $\forall x \forall y (x = y) \supset (y = x)$;
- transitivity: $\forall x \forall y \forall z ((x = y) \land (y = z)) \supset (x = z)$;
- substitution of equals for functions: for any function symbol $f$,
  \[
  \forall x_1 \ldots \forall x_k \forall y_1 \ldots \forall y_k ((x_1 = y_1) \land \cdots \land (x_k = y_k)) \supset f(x_1,\ldots,x_k) = f(y_1,\ldots,y_k);
  \]
- substitution of equals for predicates: for any predicate symbol $P$,
  \[
  \forall x_1 \ldots \forall x_k \forall y_1 \ldots \forall y_k ((x_1 = y_1) \land \cdots \land (x_k = y_k)) \supset P(x_1,\ldots,x_k) \equiv P(y_1,\ldots,y_k).
  \]
3.2 The Logic $\mathcal{L}$

Then we have the following:

**Theorem 3.14** (Levesque and Lakemeyer, [LL01]). Let $\alpha$ be a sentence of $\mathcal{L}$ without standard names. Then $\alpha$ is valid iff in classical FOL, $\Delta \cup \Gamma$ implies $\alpha$.

**Proof.** (Sketch) The proof is analogous to the one of Theorem 3.13, but where additionally $\Delta \cup \Gamma$ ensures that the correspondence between a Tarskian model $M$ and an $\mathcal{L}$ world $w$ also includes equality. Note that as opposed to FOL, the sentences in $\Delta \cup \Gamma$ are valid theorems in $\mathcal{L}$.

**Universal Generalization**

Another consequence of our usage of standard names, in addition to the substitutional interpretation of quantification, is the fact that the validity of a quantified formula can be reduced to the validity of finitely many instances. We will provide a detailed proof for this because it exemplifies a proof technique that we will make use of more often during the course of this thesis, namely proving a property of a term or formula by an induction over its structure.

**Theorem 3.15** (Universal Generalization). Let $\alpha$ be a formula with a single free variable $x$ not of sort number, and let $n$ be a standard name of the same sort as $x$ that does not appear in $\alpha$. Let $n_1, \ldots, n_k$ be all the standard names of the same sort as $x$ appearing in $\alpha$. If $\alpha_{n_i}^*$ is valid and all the $\alpha_{n_i}^*$ are valid, then so is $\forall x \alpha$.

**Proof.** Intuitively, the property holds because all names not appearing in $\alpha$ can be treated in the same manner, and hence we only need to check one such name that serves as representative for all the others. Following the proof in [LL01] for the language $\mathcal{L}$, we prove this theorem by considering a bijection $*$ from object and action standard names to object and action standard names, preserving sorts. For a term $t$, a formula $\beta$, and a sequence of action names $z$, the result of simultaneously replacing every standard name by its mapping under $*$ is denoted by $t^*, \beta^*$, and $z^*$, respectively. Further if $w$ is a world, then let $w^*$ be the world that is exactly like $w$, with the exception that for all primitive terms $t$, and action name sequences $z$,

\[(3.20) \quad w^*[t, z] = (w[t^*, z^*])^{*-1},\]

where $^{*-1}$ refers to the bijection that is the inverse of $*$. Note that $w^*$ always exists and is unique. As a consequence, we have

\[(3.21) \quad (w^*[t, z])^* = w[t^*, z^*].\]

Further let for all primitive formulas $\beta$ and action name sequences $z$,

\[(3.22) \quad w^*[\beta, z] = w[\beta^*, z^*].\]
First we show that

\[(3.23) \ (|t|_{w^*}^z)^* = |t^*|_{w^*}^z\]

for all ground terms \(t\) by an induction over their structure:

- If \(t \in N\), then \((|t|_{w^*}^z)^* = t^* = |t^*|_{w^*}^z\) by Definition 3.7.

- If \(t = f(t_1, \ldots, t_k)\), then

\[
(|t|_{w^*}^z)^* = \begin{cases} 
(w^*[f(|t_1|_{w^*}^z, \ldots, |t_k|_{w^*}^z), z])^* & \text{(by assumption (3.21))} \\
|f(t_1^*, \ldots, t_k^*)|_{w^*}^z & \text{(by definition of \(*\))} \\
|((f(t_1, \ldots, t_k)^*)|_{w^*}^z & \text{(by induction)} \\
|t^*|_{w^*}^z & \text{(by definition of \(*\))} 
\end{cases}
\]

Now we prove that for any sentence \(\beta\) and action name sequence \(z\),

\[(3.24) \quad w^*, z \models \beta \iff w, z^* \models \beta^*\]

again by induction on its structure:

- If \(\beta\) is an atom \(F(t_1, \ldots, t_k)\):

\[w^*, z \models F(t_1, \ldots, t_k) \iff \begin{cases} 
|t_1|_{w^*}^z = |t_2|_{w^*}^z & \text{(by Definition 3.8)} \\
w[F(|t_1|_{w^*}^z, \ldots, |t_k|_{w^*}^z), z] = 1 & \text{(by assumption (3.22))} \\
w[F(|t_1|_{w^*}^z, \ldots, |t_k|_{w^*}^z)^*, z] = 1 & \text{(by definition of \(*\))} \\
w[F(|t_1|_{w^*}^z, \ldots, |t_k|_{w^*}^z)^*, z] = 1 & \text{(by (3.23))} \\
w, z^* \models F(t_1^*, \ldots, t_k^*) & \text{(by Definition 3.8)} \\
w, z^* \models (F(t_1, \ldots, t_k)^*) & \text{(by definition of \(*\))} 
\end{cases}\]

- If \(\beta\) is an atom \((t_1 = t_2)\):

\[w^*, z \models (t_1 = t_2) \iff |t_1|_{w^*}^z = |t_2|_{w^*}^z & \text{(by Definition 3.8)}\]
iff \( (t_1^*_{w,v})^* = (t_2^*_{w,v})^* \) (since \( * \) is a bijection, see below)
iff \( t_1^*_{w,v} = t_2^*_{w,v} \) (by (3.23))
iff \( w, z^* \models (t_1^* = t_2^*) \) (by Definition 3.8)
iff \( w, z^* \models (t_1 = t_2)^* \) (by definition of \( * \))

Above we made use of the fact that because \( * \) is a bijection, the mapping is unique in both directions, i.e. not only does \( n_1 = n_2 \) imply that \( n_1^* = n_2^* \), but also vice versa.

- \( \beta \) has the form \( \beta_1 \land \beta_2 \):

\[
\begin{align*}
\text{iff} & \quad w^*, z \models \beta_1 \land \beta_2 \\
\text{iff} & \quad w^*, z \not\models \beta_1 \land \beta_2 \\
\text{iff} & \quad w, z^* \not\models \beta_1^* \land \beta_2^* \\
\text{iff} & \quad w, z^* \not\models \beta_1^* \land \beta_2^* \\
\text{iff} & \quad w, z^* \not\models (\beta_1 \land \beta_2)^* \\
\end{align*}
\]

- \( \beta \) has the form \( \neg \beta_1 \):

\[
\begin{align*}
\text{iff} & \quad w^*, z \not\models \neg \beta_1 \\
\text{iff} & \quad w^*, z \models \beta_1 \\
\text{iff} & \quad w, z^* \not\models \beta_1^* \\
\text{iff} & \quad w, z^* \models \neg \beta_1^* \\
\text{iff} & \quad w, z^* \models (\neg \beta_1)^* \\
\end{align*}
\]

- \( \beta \) has the form \( \forall x \beta_1 \):

\[
\begin{align*}
\text{iff} & \quad w^*, z \models \forall x \beta_1 \\
\text{iff} & \quad w^*, z \not\models \beta_1 \forall n \text{ for all } n \in N_x \\
\text{iff} & \quad w, z^* \not\models (\beta_1 \forall n)^* \text{ for all } n \in N_x \\
\text{iff} & \quad w, z^* \not\models (\beta_1 \forall n)^* \text{ for all } n \in N_x \\
\text{iff} & \quad w, z^* \not\models (\forall x \beta_1)^* \\
\text{iff} & \quad w, z^* \models (\forall x \beta_1)^* \\
\end{align*}
\]

Above we exploited the fact that because \( * \) is a sort-preserving bijection from standard names to standard names (a permutation), its image from all standard names of one sort
is again the set of all standard names of that sort, i.e.
\[ \{ n \mid n \in \mathcal{N}_x \} = \{ n^* \mid n \in \mathcal{N}_x \} = \mathcal{N}_x. \]

- \( \beta \) has the form \([t]\beta_1\):

\[
\begin{align*}
&w^*, z \models [t]\beta_1 \\
&\text{iff } w^*, z \cdot |t|^*_{w^*} \models \beta_1 \quad \text{(by Definition 3.8)} \\
&\text{iff } w, (z \cdot |t|^*_{w^*})^* \models \beta_1^* \quad \text{(by induction)} \\
&\text{iff } w, z^* \cdot (|t|^*_{w^*})^* \models \beta_1^* \quad \text{(by definition of \(*\))} \\
&\text{iff } w, z^* \cdot |t|^*_{w^*} \models \beta_1^* \quad \text{(by (3.23))} \\
&\text{iff } w, z^* \models [t]^*\beta_1^* \quad \text{(by Definition 3.8)} \\
&\text{iff } w, z^* \models ([t]\beta_1)^* \quad \text{(by definition of \(*\))}
\end{align*}
\]

- \( \beta \) has the form \(\Box\beta_1\):

\[
\begin{align*}
&w^*, z \models \Box\beta_1 \\
&\text{iff } w^*, z \cdot z' \models \beta_1 \text{ for all } z' \in \mathcal{Z} \quad \text{(by Definition 3.8)} \\
&\text{iff } w, (z \cdot z')^* \models \beta_1^* \text{ for all } z' \in \mathcal{Z} \quad \text{(by induction)} \\
&\text{iff } w, z^* \cdot z'' \models \beta_1^* \text{ for all } z'' \in \mathcal{Z} \quad \text{(by definition of \(*\))} \\
&\text{iff } w, z^* \models \Box\beta_1^* \quad \text{(by Definition 3.8)} \\
&\text{iff } w, z^* \models (\Box\beta_1)^* \quad \text{(by definition of \(*\))}
\end{align*}
\]

Above we use a similar argument as in the \(\forall\) case: Since \(*\) is a bijection from action standard names to action standard names, \(*\) applied to the set of all sequences of action names yields again the set of all action sequences:
\[
\{ z \mid z \in \mathcal{Z} \} = \{ z^* \mid z \in \mathcal{Z} \} = \mathcal{Z}.
\]

Now that we proved property (3.24), if for some \(w\) we have that \(w \not\models \beta^*\), then for \(w^*\) we have that \(w^* \not\models \beta\). Conversely, if for some \(w\) we have \(w \not\models \beta\), then since \(w\) can be expressed as \((w^{*-1})^*\), by (3.24) we get \(w^{*-1} \not\models \beta^*\). Therefore a sentence \(\beta\) is valid iff \(\beta^*\) is.

Now let \(\alpha\) have one free variable \(x\), \(n_1, \ldots, n_k\) be all standard names of the same sort as \(x\) appearing in \(\alpha\), and \(n\) be one standard name of that sort that does not appear in \(\alpha\). To show that \(\forall x\alpha\) is valid, it suffices to show that all substitution instances of \(\alpha\) are valid. In case of
3.2 The Logic $\mathcal{L}$

$n_1, \ldots, n_k, n$, the validity is already assumed. If $n'$ is a standard name distinct from $n_1, \ldots, n_k, n$, then let $*$ be the bijection that swaps $n$ and $n'$ and leaves everything else unchanged. By the above, $\alpha_{n'}^n$ is valid iff $\alpha_n^n$ is, where the latter again is assumed.

Corollary 3.16. Theorem 3.15 does not hold when the free variable is of sort number.

Proof. A counterexample is the case where $\alpha$ is the formula $m < 3 + 3$. Then $\alpha_3^m$ is the formula $3 < 3 + 3$, which is obviously valid. One number standard name not occurring in $\alpha$ is 4, and $\alpha_4^m = (4 < 3 + 3)$ is valid as well, but $\forall m.(m < 3 + 3)$ obviously is not.

For the sorts object and action we make use of the fact that standard names bear no special information and are in no specific relation among each other except for each name being distinct to any other name. The trick (first used in [Lev84]) then is that any single name not appearing in $\alpha$ can be used as a representative for all other names that do no appear, as they all behave equally in relation to the names mentioned in $\alpha$. In case of the number sort, this does obviously not work since there we actually do have special relations among the names, namely through the arithmetical operators $+, \cdot$, and $<$, whose meaning we fixed in our language’s semantics. This implies that we will not be able to reason about unknown numbers in the same way we can reason about unknown objects or actions. In particular the Representation Theorem, presented below in Section 3.5, will not be applicable either. Nonetheless, for the purpose of this thesis where only limited use of arithmetic is made, this is perfectly sufficient. In the robot scenario for instance we expect that the robot may possibly not know who is the person in the kitchen, or which food is in the refrigerator, or what action is necessary to fix the dishwasher, but it is always aware about the duration of its own actions, the distances between locations, its current energy level and the like. That is to say that the more complex reasoning tasks are to be expected within the domains of objects and actions, whereas numbers are only necessary to do simple calculations that involve numerical parameters.

3.2.4 A Remark on Numbers

Speaking of numbers, note that in our semantics we assume a fixed, countable set of numbers to serve as the standard names of sort number. At first glance this precludes the usage of the set of real numbers since this set is known to be uncountable [Can91]. On the other hand, even for the limited use of arithmetic that we want to employ in this thesis, it seems that real numbers are essential. As an example, consider that we have a robot which represents physical locations through two-dimensional coordinates. When the robot is currently at location $(0, 0)$ and an object that it wants to grab is located at $(1, 1)$, the distance between the two (which might for
instance be needed to decide whether the robot can reach the object with its robotic arm) is $\sqrt{2}$. Similarly, we may want to reason about the fuel or energy consumption of our robot, for instance to ensure that some recharging station is reachable:

$$\exists x. \text{ChargingStation}(x) \land [\text{goto(loc}(x))]\text{energy} > 0$$

When we suppose that the energy use of a $\text{goto}(x)$ action depends linearly on the travelled distance, and even in the case of simple straightline distances, this again potentially involves having to deal with irrational numbers.

One might argue that it is often not necessary to be that precise about numerical values because there is always a margin of error and because a physical computer’s internal arithmetics will approximate values such as $\sqrt{2}$ by floating point (i.e. rational) numbers anyway. However, at the logical level there is a strong reason why the agent should in some sense be “aware” that something like the square root of two exists. Consider the formula

$$x \cdot x = 2.$$ 

If we simply used an approximation of $\sqrt{2}$ such as 1.41 to substitute it for the $x$ here, we would get that $(1.41)^2 = 1.9881 = 2$, and therefore that $0.0119 = 0$, and hence also $1 = 0$ could be deduced. In other words our logic can never be consistent when we resort to such approximate values!

As discussed in [Hu06], there is however one solution that allows us to have numbers and (simple) arithmetics in our language, treat quantification over numbers substitutionally, yet include irrational numbers such as $\sqrt{2}$ and $\pi$. To understand this, we first have to take a closer look at how the real numbers are defined within mathematics.

Actually, there are several possible known constructions of the reals. While explicit approaches start from a smaller set such as the rational numbers or the integers, and build on top of that, the synthetic approach defines the real number system axiomatically [Hil00]. That is, one provides a set of axioms in the form of formulas of (classical) predicate logic such that their (any) model can be identified with the set of the reals.

More precisely, a model of the real numbers is given by a set $\mathbb{R}$, two constants 0 and 1, two operations $+: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $\cdot: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, as well as a relation $< \subseteq \mathbb{R} \times \mathbb{R}$, satisfying:

1. $(\mathbb{R}, +, \cdot)$ forms a field, i.e.

   $$(3.26) \quad \forall x, y, z \in \mathbb{R}. \ (x + y) + z = x + (y + z)$$

   $$(3.27) \quad \forall x \in \mathbb{R}. \ x + 0 = x$$

   $$(3.28) \quad \forall x \in \mathbb{R} \exists y \in \mathbb{R}. \ x + y = 0$$
3.2 The Logic $\text{ES}$

(3.29) $\forall x, y \in \mathbb{R}. x + y = y + x$

(3.30) $\forall x, y, z \in \mathbb{R}. (x \cdot y) \cdot z = x \cdot (y \cdot z)$

(3.31) $\forall x \in \mathbb{R}. x \cdot 1 = x$

(3.32) $\forall x \in \mathbb{R} \setminus \{0\} \exists y \in \mathbb{R}. x \cdot y = 1$

(3.33) $\forall x, y \in \mathbb{R}. x \cdot y = y \cdot x$

(3.34) $\forall x, y, z \in \mathbb{R}. (x + y) \cdot z = x \cdot z + y \cdot z$

(3.35) $0 \neq 1$

(3.26) to (3.29) are the rules of associativity, the neutral element, the inverse, and commutativity for addition, while (3.30) to (3.33) are the corresponding rules for multiplication. (3.34) is the distributivity law, and (3.35) expresses that the two neutral elements have to be distinct.

2. $(\mathbb{R}, <)$ is a strict total order that is compatible with $+$ and $\cdot$, i.e.

(3.36) $\forall x, y, z \in \mathbb{R}. \text{exactly one of } x < y \text{ or } x = y \text{ or } y < x \text{ holds.}$

(3.37) $\forall x, y, z \in \mathbb{R}. \text{if } x < y \text{ and } y < z \text{ then } x < z.$

(3.38) $\forall x, y, z \in \mathbb{R}. \text{if } x < y \text{ then } x + z < y + z.$

(3.39) $\forall x, y, z \in \mathbb{R}. \text{if } x < y \text{ and } 0 < z \text{ then } x \cdot z < y \cdot z.$

(3.36) is the trichotomy law, and (3.37) is the law of transitivity. (3.38) ensures the compatibility between $<$ and $+$, while (3.39) does so for $<$ and $\cdot$.

3. $\mathbb{R}$ is Dedekind-complete, i.e.

(3.40) Every bounded nonempty set $S \subseteq \mathbb{R}$ has a least upper bound.

This axiom finally ensures that the real number line is indeed continuous and does not contain any “gaps”.

In other words, axioms (3.26)–(3.40) define the real number system to be a Dedekind-complete ordered field, which is sufficient to capture all intrinsic properties of the reals. Other properties can then be derived from these axioms. Furthermore it can be shown that there is a unique model $\mathbb{R}$ in the sense that every other model $\mathbb{R}'$ has to be isomorphic, i.e. identical up to renaming and relabelling.

As indicated above, the fact that there are uncountably many reals is problematic in the context of our substitutional semantics for quantification, which requires the underlying domain
to be countable. The uncountability of the reals is due to axiom (3.40). The axiom, which is obviously second-order as it quantifies over sets of numbers, is an important property when it comes to analysis and proving theorems about the reals. However, for our purposes of doing simple arithmetical calculations, it is actually unnecessary.

Once we drop (3.40) and only keep (3.26)–(3.39), we end up with the theory of ordered fields, which may be or not be Dedekind-complete. While the real numbers are still one possible model, we now get several additional ones such as the rational numbers $\mathbb{Q}$, the algebraic numbers $\mathbb{A}$, and the computable numbers $\mathbb{C}$. Here, an algebraic number means a number that is the root of a non-zero polynomial in one variable with integer coefficients. It includes all rational numbers such as $\frac{1}{2}$ and irrational numbers like $\sqrt{2}$ and $\sqrt{4}$, but not the circle constant $\pi$ and the Euler constant $e$. The computable numbers [Tur37] are those real numbers which can be computed to any desired precision by a finite, terminating algorithm. They include all algebraic numbers and additionally important transcendent numbers such as $\pi$ and $e$. In fact we have that these sets constitute increasingly general concepts, i.e.

$$\mathbb{Q} \subset \mathbb{A} \subset \mathbb{C} \subset \mathbb{R},$$

where except for $\mathbb{R}$, each of them is countable. Note that the natural numbers $\mathbb{N}$ as well as the integers $\mathbb{Z}$ do not constitute ordered fields since they lack the inverse elements for multiplication.

Therefore, any of $\mathbb{Q}$, $\mathbb{A}$, or $\mathbb{C}$ (or any other structure that is an ordered field) is a possible candidate for our definition of the set of number standard names $\mathcal{N}_N$. For our semantics to be well defined, we just have to fix one of these sets. The arithmetical correctness constraint of Definition 3.6 is then to be understood as a mapping of the symbols $+$, $\cdot$ and $<$ to the corresponding operations within the chosen ordered field structure.

Here, we will in the following assume that $\mathcal{N}_N = \mathbb{C}$, not only because it is the largest of the above mentioned sets that is still countable, but also because it is somewhat analogous to our view of an agent that is a logically omniscient first-order reasoner. The latter means that we assume that the agent is always aware of all logical consequences of the formulas in its knowledge base, despite the fact that any physical implementation will suffer certain computability limitations due to the semi-decidability of first-order logic. Similarly, in our idealized perspective the exact result of any arithmetical calculation is always instantly available to the agent, although in practice one would have to resort to approximations whose accuracy depends on the computation time invested. Consequently, we will in the following make free use of the elementary operations $+$, $\cdot$, and $<$ within formulas and view number constants such as $42$, $\frac{1}{2}$, $\sqrt{2}$ and $\pi$ as number standard names. Furthermore, expressions involving the inverse
3.2 The Logic $\mathcal{LS}$

operations − (subtraction) or / (division) will be viewed as abbreviations in terms of + and ·:

\[(x - y = z) \stackrel{def}{=} (y + z = x)\]
\[(x/y = z) \stackrel{def}{=} (y \cdot z = x)\]

3.2.5 Second-Order Quantification

Sometimes it may be necessary for us to be able to quantify not only over single individuals, but also over sets, relations and functions. As mentioned earlier, we assume that the agent itself only possesses first-order reasoning capabilities. Second-order quantification may however be necessary on the one hand to express facts about the agent and its environment from a meta-perspective, and on the other hand to point out the limits of what is representable by the sole usage of first-order formulas. The macro-definition of the GOLOG semantics presented in Section 3.6 is an example for the former, the second-order LR-Progression discussed in Section 3.4.2 for the latter.

To include second-order quantification, we extend our language’s syntax as follows:

**Definition 3.17.** The set of objective SO formulas is constructed analogously to Definition 3.2, with the additional rule:

4. If $\alpha$ is an objective SO formula, $P$ a (fluent or rigid) predicate symbol, and $f$ a (fluent or rigid) function symbol, then $\forall P.\alpha$ and $\forall f.\alpha$ are objective SO formulas.

For interpreting second-order quantification semantically, we define a compatibility relation among worlds wrt some predicate or function symbol:

**Definition 3.18.** Let $w$ and $w'$ be worlds, $P$ a (fluent or rigid) predicate symbol, $f$ a (fluent or rigid) function symbol, and $z \in Z$. Then $w \sim_P w'$ iff for all primitive sentences $P' (\vec{n})$ with $P' \neq P$ and all $z' \in Z$,

\[w[P'(\vec{n}), z \cdot z'] = w'[P(\vec{n}), z \cdot z'].\]

Similarly, $w \sim_f w'$ iff for all primitive terms $f' (\vec{n})$ with $f' \neq f$ and all $z' \in Z$,

\[w[f'(\vec{n}), z \cdot z'] = w'[f(\vec{n}), z \cdot z'].\]

Intuitively, the definition says that $w \sim_P w'$ iff $w$ and $w'$ are identical from $z$ onwards, except perhaps on their interpretation of primitive formulas involving $P$. Similarly, $w \sim_f w'$ iff $w$ and $w'$ are identical from $z$ onwards, except perhaps on their interpretation of primitive terms involving $f$. If $z$ is the empty sequence, we also write $\sim_P$ and $\sim_f$ instead of $\sim_P$ and $\sim_f$, respectively.
Proposition 3.19. For any $P$, $f$ and $z$, $\sim_P$ and $\sim_f$ are equivalence relations.

Proposition 3.20. If $w' \sim_P w$, then $w' \sim_P w$.

We will further employ an extended notation that allows $\sim$ to be used with multiple predicates and/or functions. $w \sim_{P_1, P_2, f_1} w'$ then for instance means that from $z$ onwards, $w$ and $w'$ agree on every primitive formula that mentions neither $P_1$ nor $P_2$ and on every primitive term without $f_1$.

Proposition 3.21. If $w' \sim_{P_1} w$, then $w' \sim_{P_1, P_2} w$.

The semantics of formulas with second-order quantifiers is now given by the definition below:

Definition 3.22. For objective SO formulas, we extend Definition 3.8 by the following additional rules:

8. $w, z \models \forall P. \alpha$ iff $w', z \models \alpha$ for all $w' \sim_P w$;

9. $w, z \models \forall f. \alpha$ iff $w', z \models \alpha$ for all $w' \sim_f w$.

Note that the above definition of second-order ES slightly differs from the one in the existing literature such as [LL05a]. Whereas Lakemeyer and Levesque assume that the logic’s alphabet contains a separate pool of uncountably many second-order variable symbols, we simply allow to quantify any of the already available function and predicate symbols. That is to say that the role of any such symbol differs between being a second-order variable or a normal non-logical symbol, depending on whether it is within the scope of a corresponding second-order quantifier or not. In consequence, we do not need to include special second-order variable maps in our semantics as in [LL05a], but it suffices to let worlds interpret functions and predicates as before. Semantically, a second-order quantifier $\forall P$ here then does not range over all possible second-order variable maps, but over all worlds with different extensions of $P$.

A typical application of second-order quantification is the definition of the transitive closure of a given relation. Consider the $In$ predicate that we used in our robot example. When we have facts telling us that Bob is in his office and Bob’s office is in the building of the Computer Science Department, i.e.

\[(3.43)\quad In(bob, office(bob)) \land In(office(bob), csbuilding),\]

we would like to be able to deduce that this also means that Bob is currently inside the Computer Science building, without explicitly including this fact in our theory. The fact that the binary relation $In$ is transitive can easily be expressed as follows:

\[(3.44)\quad \forall x \forall y \forall z. In(x, y) \land In(y, z) \supset In(x, z)\]
Unfortunately, this sentence is not sufficient to capture the notion of a transitive closure. The reason is that it is nowhere said that \( In \) has to be interpreted by the smallest set that satisfies the transitivity rule. In fact, the full Cartesian product of all pairs of object standard names would be a model, i.e. any \( w \) such that \( w[In(n_1, n_2), z] = 1 \) for all \( n_1 \in N_O \), all \( n_2 \in N_O \), and all \( z \in Z \) satisfies (3.44). As a consequence, \( In(csbuilding, office(bob)) \) and \( In(csbuilding, interior(moon)) \) cannot be excluded, contrary to our intuition.

It is possible though to define the transitive closure of \( In \) by means of second-order quantification. Let \( \alpha \) denote (3.43) and \( \beta \) mean (3.44). Then take the sentence

\[
InTrans(x', y') \equiv \forall In(\alpha \land \beta \supset In(x', y'))
\]

saying that the true instances of \( InTrans(x', y') \) are exactly those pairs \( \langle x', y' \rangle \) such that \( In(x', y') \) is true in all possible interpretations of \( In \) satisfying (3.43) and (3.44). The idea is then what holds for all interpretations must also be true for the smallest one, which then corresponds to the actual transitive closure. Note that (3.45) now correctly entails both \( InTrans(bob, csbuilding) \) and \( \neg InTrans(csbuilding, office(bob)) \).

### 3.3 Basic Action Theories

Now that we have defined the fundamental part of the language \( ES \) (later we will discuss a number of extensions), we are ready to use it in order to represent the dynamics of specific application domains. Typically, for any such domain this involves at least a description of

1. what the world is currently like,
2. the preconditions of the actions at the agent’s disposal and
3. how the agent’s actions affect the state of the world.

The first item does not pose a big problem. In order to describe what is currently true, no actions or modal operators are required, but standard first-order syntax is sufficient. For example, in the case of our office robot, a simple first-order sentence like \( \neg \exists x.Holding(x) \) can be used to express that the robot is currently not holding anything in its robotic arm.

Each of the other two items however comes with a problem that has been recognized early and studied thoroughly in the KR literature. When describing the action’s preconditions, we are confronted with the so-called qualification problem, and in the case of their postconditions (effects), we have to deal with the frame problem. We will present these problems together with possible (simple) solutions in the following.
3.3.1 The Qualification Problem

*Preconditions* of an action are requirements that have to be met in order for the agent to be able to execute that action in the current situation. To formalize such requirements through logical formulas we use the fluent predicate $\text{Poss}(a)$ whose intended meaning is that it is true whenever the argument $a$ is an action that is possible to execute.

As shown in [Rei01a], it is normally straightforward to write down a collection of necessary conditions for a given action. For instance, consider the action $\text{pickup}(y)$: In order for the robot to be able to pick up an object $y$, it should not be holding anything else currently, it should be at the same location as the object, and the object should not be too heavy (we imagine a robot that has only one robotic arm and therefore can only hold one object at a time):

\[
\Box \text{Poss}(\text{pickup}(y)) \supset \neg \exists x \text{Holding}(x) \tag{3.46}
\]
\[
\Box \text{Poss}(\text{pickup}(y)) \supset \text{loc}(y) = \text{loc}(\text{robot}) \tag{3.47}
\]
\[
\Box \text{Poss}(\text{pickup}(y)) \supset \neg \text{Heavy}(y) \tag{3.48}
\]

All conditions of this form can be easily combined into a single sentence by means of conjunction:

\[
\Box \text{Poss}(\text{pickup}(y)) \supset \neg \exists x \text{Holding}(x) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \text{Heavy}(y) \tag{3.49}
\]

The problem now is that while this formula certainly expresses true facts about the executability of a $\text{pickup}(y)$ action, by itself it is actually useless. The reason is that from (3.49) alone we can at most infer when $\text{pickup}(y)$ is not possible, but not when it is. What is missing in addition to the necessary conditions given in the axiom is that we also explicitly state all *sufficient* conditions. The *qualification problem* now refers to the difficulty of foreseeing every possible such condition. As an example, it is not sufficient to simply reverse the implication of (3.49):

\[
\Box \neg \exists x \text{Holding}(x) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \text{Heavy}(y) \supset \text{Poss}(a) \tag{3.50}
\]

We would rather have to include also something along the lines of the following:

\[
\neg \text{GluedToFloor}(y) \land \neg \text{ArmsTied} \land \neg \text{HitByTenTonTruck} \land \ldots \tag{3.51}
\]

Even in the improbable case that we are able to come up with a complete list of all these “minor” qualifications, this is problematic. When all the agents knows is that

\[
\neg \exists x \text{Holding}(x) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \text{Heavy}(y) \tag{3.52}
\]

holds and it is not known to it whether any of the “minor” conditions is true, then it still cannot be inferred whether $\text{pickup}(y)$ is possible, at least by means of ordinary monotonic logics. What
would be needed here is a form of nonmonotonic reasoning such as Reiter’s Default Logic [Rei80],
which uses rules of the form
\[ \alpha : \beta_1, \ldots, \beta_k \rightarrow \gamma \]

The intuition is that when the prerequisite \( \alpha \) is known to hold and none of the justifications \( \beta_1, \ldots, \beta_k \) is known to be false, then by default, infer the conclusion \( \gamma \). In our case, the prerequisite would be the “major” qualifications (3.52), the justifications are given by the “minor” qualifications (3.51), and the conclusion is \( \text{Poss}(\text{pickup}(y)) \).

For the purpose of this thesis, we simply decide to basically ignore the “minor” qualifications and only deal with the “major” ones. We should though keep in mind that this is again an idealization, and that practical implementations require some mechanism that enables the system to cope with unforeseen events or unexpected situations.

In general, we assume that for each action operator \( g_j(y_j) \), we have one axiom stating the necessary (“major”) preconditions:
\[
\square \text{Poss}(g_1(y_1)) \supset \pi_1 \\
\ldots \\
\square \text{Poss}(g_l(y_l)) \supset \pi_l
\]

Each \( \pi_j \) is a formula without any modal operators and whose free variables are the corresponding \( y_j \). Furthermore, there may be one general axiom that does not depend on a particular operator:
\[
\square \text{Poss}(a) \supset \pi_a
\]

Again \( \pi_a \) does not contain modal operators, and its only free variable is \( a \). An example for a qualification axiom of this type in the robot scenario could be the precondition that no matter what action is to be performed, the robot must have some energy left:
\[
\square \text{Poss}(a) \supset \text{energy} > 0 \tag{3.53}
\]

We then make an idealized completeness assumption that these axioms in fact encode all the necessary and sufficient conditions for an action \( a \) to be possible. A single precondition axiom can then be constructed as follows:
\[
\square \text{Poss}(a) \equiv (\bigvee_{j=1}^l \exists y_j (a = g_j(y_j) \land \pi_j)) \land \pi_a \tag{3.54}
\]

In the example, we might have the following axioms for the operators \( \text{goto}(y) \), \( \text{pickup}(y) \), \( \text{putdown}(y) \), and \( \text{recharge} \), respectively:
\[
\square \text{Poss}(\text{goto}(y)) \supset \text{Location}(y) \land \text{NextTo}(\text{loc(robot)}, y) \tag{3.55}
\]
\(\square \text{Poss}(\text{pickup}(y)) \supset \neg \exists x \text{Holding}(x) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \text{Heavy}(y)\)

\(\square \text{Poss}(\text{putdown}(y)) \supset \text{Holding}(y)\)

\(\square \text{Poss}(\text{recharge}) \supset \exists x (\text{ChargingStation}(x) \land \text{loc}(x) = \text{loc}(\text{robot}))\)

That is, going to \(y\) requires \(y\) to be at a location that is next to the robot’s one. Picking up an object is possible only if the robot is not holding anything else, it is at the object’s position, and the object is not too heavy. An object can be put down only if it is currently being held, and recharging is possible only when the robot is at a location where there is some recharging station. Let (3.53) be the operator independent qualification axiom. The precondition axiom then is:

\(\square \text{Poss}(a) \equiv \)
\[
\{ \exists y(a = \text{goto}(y) \land \text{Location}(y) \land \text{NextTo}(\text{loc}(\text{robot}), y)) \lor \\
\exists y(a = \text{pickup}(y) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \exists x \text{Holding}(x) \land \neg \text{Heavy}(y)) \lor \\
\exists y(a = \text{putdown}(y) \land \text{Holding}(y)) \lor \\
(a = \text{recharge} \land \exists x (\text{ChargingStation}(x) \land \text{loc}(x) = \text{loc}(\text{robot}))) \}
\]
\(\land \text{energy} > 0\)

### 3.3.2 The Frame Problem

It is also comparatively easy to come up with formulas that describe what direct effects actions have on the state of a fluent. For instance, picking up an object will cause it to be held by the robot, while putting it down has the effect that the object is not held anymore:

\(\square [\text{pickup}(x)] \text{Holding}(x)\)

\(\square [\text{putdown}(x)] \neg \text{Holding}(x)\)

We call formulas of this form effect axioms. These axioms by themselves are however not sufficient to fully describe the new state that results from doing some action. The reason is that effect axioms only encode the changes that are caused by the actions, but they do not encode what is left unchanged. For example, the above effect axiom (3.60) does not exclude the possibility that the colour of the object changes while picking it up:

\(\{ \text{colour}(x) = y, [\text{pickup}(x)] \text{Holding}(x) \} \not\models [\text{pickup}(x)] \text{colour}(x) = y\)

This is of course counter-intuitive as one would expect the colour of an object to remain unaffected by a pickup action. To correctly account for such invariants, one would have to resort to
so-called frame axioms. In the above example, the corresponding frame axiom would be

\[(3.63) \quad \square \text{colour}(x) = y \supset [\text{pickup}(x)] \text{colour}(x) = y.\]

The problem now is the sheer number of such axioms that one would have to include in the theory. Picking up an object for instance not only leaves the colour of that object unchanged, but also the colour of every other object, the positions of other objects, the positions of people in the building, whether doors are open or closed, and so on. In general, if \(N_A\) is the number of actions to be considered, and \(N_F\) the number of fluents, then the total number of effect and frame axioms would be around \(2 \cdot N_A \cdot N_F\).

Listing all effects and non-effects for all possible combinations of actions and fluents in this way is not only tedious and error-prone, but it is also somewhat redundant. Once one has come up with a complete list of all actions’ effects, that list already contains all necessary information. What it however is missing is a logical representation of the fact that the listed effects comprise all possible effects and that in any other case, by default, no change happens. Furthermore, because actions usually only affect a few fluents, there is only a small number of effect axioms, but a huge number of frame axioms would be needed.

Reiter’s popular solution [Rei91], which is based on earlier proposals by Pednault [Ped89], Davis [Dav90], Haas [Haa87], and Schubert [Sch90], is as follows. Given a set of relevant fluents \(\mathcal{F}\), we assume that for each relational \(F \in \mathcal{F}\), we have exactly one positive effect axiom of the form

\[(3.64) \quad \square \gamma_F^+ \supset [a]F(\vec{x}).\]

and one negative effect axiom of the form

\[(3.65) \quad \square \gamma_F^- \supset [a] \neg F(\vec{x}).\]

\(\gamma_F^+\) and \(\gamma_F^-\) have to be formulas that do not contain any modal operators and whose only free variables are \(\vec{x}\) and \(a\). Here, \(\gamma_F^+\) describes all conditions under which \(F(\vec{x})\) is caused to become true by executing action \(a\), while \(\gamma_F^-\) gives the conditions under which \(a\) makes \(F(\vec{x})\) false. The above represents a normal form to which our example effect axioms (3.60) and (3.61) above do not yet conform. However, they can easily brought into this form by means of equalities:

\[(3.66) \quad \square (a = \text{pickup}(x)) \supset [a] \text{Holding}(x)\]

\[(3.67) \quad \square (a = \text{putdown}(x)) \supset [a] \neg \text{Holding}(x)\]
Furthermore, in general one may have more than one effect axiom of a sort, for example when there are multiple actions which can possibly affect a fluent $F$. A collection of $k$ axioms

$$
\square \psi_F^{(i)} \supset [a]F(\vec{x}) \\
\vdots \\
\square \psi_F^{(k)} \supset [a]F(\vec{x})
$$

can be combined into a single one by using a disjunction:

$$
\square (\psi_F^{(1)} \lor \cdots \lor \psi_F^{(k)}) \supset [a]F(\vec{x})
$$

This construction moreover captures the case where we do not have any positive effect axiom for $F$, e.g. when there is no action whatsoever that can potentially cause $F$ to become true. Following the usual convention, a disjunction over an empty set is interpreted as “false”, and thus we get in this case the effect axiom

$$
\square \bot \supset F(\vec{x}).
$$

All of the above of course can be similarly applied to negative effect axioms. Now, given a single positive effect axiom and a single negative one for each relational fluent $F$, one makes the causal completeness assumption: Axioms (3.64) and (3.65), respectively, characterize all the conditions under which action $a$ causes $F$ to become true (respectively, false) in the successor situation. In other words, when $F$’s truth value changes from true to false by action $a$, then $\gamma_F^-$ must hold, and similarly, if $F$ changes from false to true, this is due to $\gamma_F^+$ being true. Formally, this is expressed by the explanation closure axioms:

(3.68)  $$
\Box F(\vec{x}) \land [a] \neg F(\vec{x}) \supset \gamma_F^-
$$

(3.69)  $$
\Box \neg F(\vec{x}) \land [a] F(\vec{x}) \supset \gamma_F^+
$$

**Theorem 3.23** (Reiter, [Rei91]). *Suppose $\Gamma$ is a background theory that entails*

(3.70)  $$
\Box \neg (\gamma_F^+ \land \gamma_F^-).
$$

*Then $\Gamma$ entails that the effect axioms (3.64) and (3.65), together with the explanation closure axioms (3.68) and (3.69), are logically equivalent to:*

(3.71)  $$
\Box [a] F(\vec{x}) \equiv \gamma_F^+ \lor F(\vec{x}) \land \neg \gamma_F^-
$$
We call (3.71) a successor state axiom (SSA). The intuition is that \( F(\vec{x}) \) is true after doing action \( a \) if it was caused to become true, or it was already true and not caused to turn false. In the example of the \( Holding \) fluent, the corresponding successor state axiom thus is

\[
\Box[a]Holding(x) \equiv (a = \text{pickup}(x)) \lor Holding(x) \land (a \neq \text{putdown}(x)).
\]

Reiter’s is an extremely elegant solution to the frame problem as instead of the \( 2 \cdot N_A \cdot N_F \) many effect and frame axioms, now only \( N_F \) successor state axioms are needed. One may object to this by pointing out that SSAs are larger in size than effect axioms because the former are constructed out of several of the latter. However, as each effect axiom is only used within a single SSA, the total size of the SSAs corresponds roughly to the total size of all effect axioms. This together with the fact that in a typical application, each action only affects a small, limited number of fluents, while a much larger number of fluents remains unaffected, implies that SSAs comprise a much more compact representation. Moreover, SSAs are also the basis for regression-based reasoning, which we will discuss in Section 3.4.1.

The consistency condition (3.70) is necessary to ensure that a fluent is not caused to be true and false at the same time, which obviously leads to a contradiction. For the fluent \( Holding \), the condition is satisfied because of unique names for actions. Reiter’s solution for the original Situation Calculus indeed included the unique names assumption for actions as an explicit requirement. In the case of \( ES \) however, this is unnecessary as we get it “for free” by the language’s semantics, cf. Proposition 3.9.

For functional fluents, successor state axioms can be constructed in a similar manner. The assumption here is that for any functional \( f \in \mathcal{F} \), we have one effect axiom of the form

\[
\Box[γ_f^+] \supset [a] (f(\vec{x}) = y),
\]

where \( γ_f^+ \) is a subformula without any modal operators and whose free variables are among \( \vec{x}, y, \) and \( a \). Note that we do not have a negative effect axiom since functions cannot “lose” their value as the logic’s semantics defines them to be total.

Similar to the relational case, we now have to make a causal completeness assumption, meaning that we assume that (3.73) characterizes all the conditions under which action \( a \) can cause \( f \) to take value \( y \) in the successor situation. Formally, this is expressed again by means of an explanation closure axiom:

\[
\Box \exists y' \exists y'' ((f(\vec{x}) = y') \land [a] (f(\vec{x}) = y'') \land (y' \neq y'')) \supset \exists y γ_f^+
\]

This axiom says that when \( f \) has a different value before and after executing action \( a \), then this is due to action \( a \) causing it, under the conditions expressed in \( γ_f^+ \). Just as in the relational
case, assuming the consistency condition

\[(3.75)\quad \Box \neg \exists y \exists y'. \gamma_f^+ \land \gamma^+_f y \land (y \neq y')\]

it can then be shown that (3.73) together with (3.74) is equivalent to

\[(3.76)\quad \Box [a](f(x) = y) \equiv (f(x) = y) \land \neg \exists y' \gamma_f^+ y\]

which we call the successor state axiom for functional fluent \( f \).

As an example, consider the fluent \( \text{loc}(x) \), denoting the location of an individual \( x \), where \( x \) can refer to a physical object, but also to the robot or a person. The location of an object \( x \) changes to \( y \) by doing a \( \text{goto}(y) \) action in case \( x \) is the robot itself or if it is an object held by the robot:

\[(3.77)\quad \Box (a = \text{goto}(y) \land x = \text{robot}) \supset [a](\text{loc}(x) = y)\]

\[(3.78)\quad \Box (a = \text{goto}(y) \land \text{Holding}(x)) \supset [a](\text{loc}(x) = y)\]

Using disjunction and simplification, the two effect axiom can be combined into a single one:

\[(3.79)\quad \Box (a = \text{goto}(y) \land (x = \text{robot} \lor \text{Holding}(x))) \supset [a](\text{loc}(x) = y)\]

The successor state axiom then is

\[(3.80)\quad \Box [a](\text{loc}(x) = y) \equiv \]

\[a = \text{goto}(y) \land (x = \text{robot} \lor \text{Holding}(x)) \lor \]

\[\text{loc}(x) = y \land \neg \exists y'(a = \text{goto}(y') \land (x = \text{robot} \lor \text{Holding}(x))).\]

Note that again the consistency condition (3.75) is satisfied because unique names for actions ensure that \( a = \text{goto}(y) \) can only hold for one standard name at a time.

### 3.3.3 Formal Definition

In later parts of this thesis we will introduce further special fluents for which we use axioms similar to the precondition axiom that defines \( \text{Poss} \). In the following, we therefore distinguish two types of fluents: On the one hand, there is a set of normal fluents \( F \) for which we have successor state axioms, and on the other hand there is a set of definitional fluents \( D \) whose truth values are defined in terms of the values of normal fluents in the same situation.

**Definition 3.24** (Fluent Formulas and Terms). Let \( D \) and \( F \) be two sets of (functional and relational) fluents such that \( D \cap F = \emptyset \). An objective formula \( \phi \) is fluent wrt \( \langle D, F \rangle \) if
• $\phi$ is static,
• $\phi$ does not mention any fluents from $\mathcal{D}$ and
• all fluents mentioned in $\phi$ are from $\mathcal{F}$.

Similarly, we say that a term is fluent when it does not contain any function symbols $d \in \mathcal{D}$ and all functional fluents that it does contain are from $\mathcal{F}$.

We often omit the phrase “wrt $\langle \mathcal{D}, \mathcal{F} \rangle$” when the two sets of fluents are understood.

**Definition 3.25.** Let $\langle \mathcal{D}, \mathcal{F} \rangle$ be sets of (functional and relational) fluents such that $\mathcal{D} \cap \mathcal{F} = \emptyset$. A basic action theory over $\langle \mathcal{D}, \mathcal{F} \rangle$ (BAT) is a set of sentences of the form

$$\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}},$$

where

• $\Sigma_0$ is a set of fluent sentences wrt $\langle \mathcal{D}, \mathcal{F} \rangle$ called the initial database.

• $\Sigma_{\text{def}}$ is the definitional part and consists of one sentence of the form

$$\Box D(\vec{x}) \equiv \varphi_D \quad (3.81)$$

for each relational $D \in \mathcal{D}$, where $\varphi_D$ is a fluent formula wrt $\langle \mathcal{D}, \mathcal{F} \rangle$ whose free variables are $\vec{x}$, and one sentence

$$\Box (d(\vec{x}) = y) \equiv \varphi_d \quad (3.82)$$

for each functional $d \in \mathcal{D}$, where $\varphi_d$ is a fluent formula wrt $\langle \mathcal{D}, \mathcal{F} \rangle$ whose free variables are $\vec{x}$ and $y$.

• $\Sigma_{\text{post}}$ is a set of successor state axioms

$$\Box [a]F(\vec{x}) \equiv \gamma_F \quad (3.83)$$

for each relational $F \in \mathcal{F}$, where $\gamma_F$ is a fluent formula wrt $\langle \mathcal{D}, \mathcal{F} \rangle$ with free variables $\vec{x}$ and $a$, and

$$\Box [a](f(\vec{x}) = y) \equiv \gamma_f \quad (3.84)$$

for each functional $f \in \mathcal{F}$, where $\gamma_f$ is a fluent formula wrt $\langle \mathcal{D}, \mathcal{F} \rangle$ with free variables $\vec{x}$, $y$, and $a$. 

Typically, we have at least one definitional fluent in the form of the $\text{Poss}(a)$ predicate. The axiom

$$\square \text{Poss}(a) \equiv \varphi_{\text{Poss}}$$

then is the so-called *precondition axiom* that defines the sufficient and necessary conditions under which actions $a$ are executable. Later, we will introduce additional definitional fluents when needed, e.g. for sensing results or exogenous actions.

**Example 3.26.** Let us take a look at what a basic action theory might look like in our robot example. An initial database could consist of the following sentences:

\begin{align*}
(3.85) & \quad \neg \exists x. \text{Holding}(x) \\
(3.86) & \quad \text{energy} = 50 \\
(3.87) & \quad \text{loc(robot)} = \text{hallway} \\
(3.88) & \quad \text{loc(cup)} = \text{kitchen} \\
(3.89) & \quad \neg \text{Heavy(cup)} \\
(3.90) & \quad \text{NextTo}(\text{office(ann)}, \text{hallway}) \land \text{NextTo}(\text{office(bob)}, \text{hallway}) \\
& \quad \land \quad \text{NextTo}(\text{kitchen}, \text{hallway}) \\
(3.91) & \quad \forall x \forall y. \text{NextTo}(x, y) \supset \text{NextTo}(y, x) \\
(3.92) & \quad \text{distance}(\text{office(ann)}, \text{hallway}) = 5 \land \text{distance}(\text{office(bob)}, \text{hallway}) = 3 \\
& \quad \land \quad \text{distance}(\text{kitchen}, \text{hallway}) = 4 \\
(3.93) & \quad \forall x \forall y. \text{distance}(x, y) = \text{distance}(y, x) \\
(3.94) & \quad \exists x. \text{ChargingStation}(x) \land \text{loc}(x) = \text{office(bob)} \\
(3.95) & \quad \text{Person}(\text{ann}) \land \text{Person}(\text{bob}) \\
(3.96) & \quad \text{Room}(\text{office(ann)}) \land \text{Room}(\text{office(bob)}) \land \text{Room}(\text{hallway}) \\
(3.97) & \quad \forall x. \text{Room}(x) \supset \text{Location}(x)
\end{align*}

That is initially, the robot is not holding anything (3.85), its energy level is at 50 percent (3.86), and it is located in the hallway (3.87). The cup of coffee is initially in the kitchen (3.88), and it is not heavy (3.89). Furthermore, each of Ann’s office, Bob’s office and the kitchen is connected to the hallway (3.90), where the distances are as specified in (3.92). Both $\text{NextTo}$ and $\text{distance}$ are symmetric (3.91), (3.93). There is a charging station in Bob’s office (3.94).

The only definitional fluent here is the $\text{Poss}$ predicate, i.e. $\mathcal{D} = \{\text{Poss}\}$. The definitional
part $\Sigma_{\text{act}}$ therefore solely consists of the precondition axiom (3.59), repeated below:

$\BoxPoss(a) \equiv$

\[
\{ \exists y (a = \text{goto}(y) \land Location(y) \land \text{NextTo}(\text{loc}(\text{robot}), y)) \lor \\
\exists y (a = \text{pick}(y) \land \text{loc}(y) = \text{loc}(\text{robot}) \land \neg \exists x \text{Holding}(x) \land \neg \text{Heavy}(y)) \lor \\
\exists y (a = \text{put}(y) \land \text{Holding}(y)) \lor \\
(a = \text{recharge} \land \exists x (\text{ChargingStation}(x) \land \text{loc}(x) = \text{loc}(\text{robot}))) \}
\]

$\land \text{energy} > 0$

The axiom says that a goto action is possible iff the destination is adjacent to the robot’s current location. Picking up an object is possible when the object is at the robot’s location and it is currently not holding anything else. An object can be put down if it is currently held, and the robot can recharge if there is a recharging station at its location. In any case the robot has to have some energy left to do an action.

The normal fluents in this example are $\mathcal{F} = \{ \text{loc}, \text{Holding}, \text{energy}, \text{distance} \}$. We have one successor state axiom for each of them:

\[
\Box[a](\text{loc}(x) = y) \equiv \\
(a = \text{goto}(y) \land (x = \text{robot} \lor \text{Holding}(x)) \lor \\
\text{loc}(x) = y \land \neg \exists y'(a = \text{goto}(y') \land (x = \text{robot} \lor \text{Holding}(x)))
\]

\[
\Box[a]\text{Holding}(x) \equiv \\
(a = \text{pick}(x) \lor \neg \text{Holding}(x) \land a \neq \text{put}(x)
\]

\[
\Box[a](\text{energy} = y) \equiv \\
\exists x (a = \text{goto}(x) \land y = \text{energy} - 3 \cdot \text{distance}(x, \text{loc}(\text{robot}))) \lor \\
\exists x (a = \text{pick}(x) \land y = \text{energy} - 10) \lor \\
\exists x (a = \text{put}(x) \land y = \text{energy} - 10) \lor \\
(a = \text{recharge} \land y = 100)
\]

\[
\Box[a](\text{distance}(x_1, x_2) = y) \equiv \\
\text{distance}(x_1, x_2) = y
\]

The location of an object $x$ changes to $y$ iff $x$ is the robot and it moves to $y$, or if is an object held by the robot while moving to $y$, or if it is already at $y$ and is not moved elsewhere (3.99). The robot will hold an object $x$ after picking it up or when it is already holding it and does not put it down (3.100). The energy costs of moving from one location to the other are three times the travelled distance in percentage points, and those of grabbing an object or putting it down
amount to ten units. The recharge operation sets the energy level back to one hundred (3.101). Finally, since we defined all numeric functions to be fluent, we have one SSA to assert that distances do not change by the robot’s actions (3.102). Note that we do not have any successor state axioms for Heavy, Location, NextTo and ChargingStation as these are rigid predicate symbols. If Σ is the above basic action theory, a few simple consequences are as follows:

(3.103) \( \Sigma \models \neg \text{Holding}(\text{cup}) \)
(3.104) \( \Sigma \not\models \text{Poss}(\text{pickup}(\text{cup})) \)
(3.105) \( \Sigma \models \text{Poss}(\text{goto}(\text{loc}(\text{cup}))) \)
(3.106) \( \Sigma \models [\text{goto}(\text{loc}(\text{cup}))] \text{loc}(\text{robot}) = \text{kitchen} \)
(3.107) \( \Sigma \models [\text{goto}(\text{loc}(\text{cup}))] \text{energy} = 38 \)
(3.108) \( \Sigma \models [\text{goto}(\text{loc}(\text{cup}))] \text{Poss}(\text{pickup}(\text{cup})) \)
(3.109) \( \Sigma \models [\text{goto}(\text{loc}(\text{cup}))][\text{pickup}(\text{cup})] \text{Holding}(\text{cup}) \)

### 3.4 Projection

Entailments such as the ones given in the above example are instances of what is called the projection problem. This term refers to probably the most important reasoning task for an autonomous, knowledge-based agent, which is to decide whether some formula \( \phi \) will hold after executing a sequence of ground actions \( \langle g_1(\vec{t}_1), \ldots, g_k(\vec{t}_k) \rangle \), given some background theory \( \Sigma \):

(3.110) \( \Sigma \models [g_1(\vec{t}_1)] \cdots [g_k(\vec{t}_k)] \phi \)

Obviously, projection is needed in the context of planning. If \( \langle g_1(\vec{t}_1), \ldots, g_k(\vec{t}_k) \rangle \) is one candidate plan, then it needs to be checked whether it achieves the goal formula \( \phi \). Furthermore, it needs to be ensured that in each situation along such a plan, the next action to be executed is indeed executable according to Poss. These executability checks again have the form (3.110), where \( \langle g_1(\vec{t}_1), \ldots, g_k(\vec{t}_k) \rangle \) in this case is a partial plan leading to the current situation and \( \phi \) refers to the precondition of the subsequent action \( g_{k+1}(\vec{t}_{k+1}) \).

Projection can be solved in two possible directions. On the one hand one can transform the query \( \phi \) backwards through actions \( \langle g_1(\vec{t}_1), \ldots, g_k(\vec{t}_k) \rangle \) into an equivalent formula \( \phi' \) about the initial situation. The problem can then be solved by deciding whether \( \Sigma_0 \models \phi' \), which does not involve actions anymore and can therefore be done by standard first-order theorem proving.

On the other hand, it is also possible to leave the query \( \phi \) unchanged, but transform the initial database \( \Sigma_0 \) forwards into an equivalent theory \( \Sigma'_0 \) that is updated by the changes in-
duced through actions \(\langle g_1(\vec{t}_1), \ldots, g_k(\vec{t}_k)\rangle\). The projection problem thus reduces to the question whether \(\Sigma_0 \models \phi\), which can again be decided by standard first-order reasoning alone.

The former approach is called regression, while the latter is referred to as progression. Both have their advantages and disadvantages, and we will discuss those along with formal definitions in the following.

### 3.4.1 Regression

The successor state axioms that result from Reiter’s solution to the frame problem allow for a particular elegant solution to the projection problem called regression. The underlying idea is explained quickly: Consider a non-static formula like

\[(3.111) \quad \text{[pickup}(\text{cup})]\text{Holding}(\text{cup}),\]

expressing that the robot will be holding the \text{cup} after doing action \text{pickup}(\text{cup}). Now the successor state axiom for the \text{Holding} fluent is

\[\square[a]\text{Holding}(x) \equiv a = \text{pickup}(x) \lor \lnot\text{Holding}(x) \land a \neq \text{putdown}(x),\]

which tells us that for all \(x\), in any situation and after any action \(a\), \text{Holding}(x) is true iff the right-hand side of the axiom holds before doing \(a\). When we instantiate (i.e. substitute) \(a\) by \text{pickup}(\text{cup}) and \(x\) by \text{cup}, this implies that according to the successor state axiom, \((3.111)\) is equivalent to

\[(3.112) \quad \text{pickup}(\text{cup}) = \text{pickup}(\text{cup}) \lor \lnot\text{Holding}(\text{cup}) \land \text{pickup}(\text{cup}) \neq \text{putdown}(\text{cup}).\]

We may therefore replace the original formula by this one, preserving equivalence wrt the basic action theory. The nice thing about this is that this new formula now does not contain any \([t]\) operators. In general, each regression step removes one such operator, and after a finite number of substitutions, we end up with a static formula. The result can then simply be checked against the initial database \(\Sigma_0\), and since no modal operators are involved anymore, standard first-order reasoning suffices. In the above example, formula \((3.112)\) is trivially entailed by any \(\Sigma_0\) since it is obviously valid.

Similarly, whenever a formula contains an atom Poss\((t)\), we can simply replace it by the right-hand side of our precondition axiom, where the variable \(a\) needs to be substituted by the term \(t\) mentioned by the atom. Accordingly we can proceed for any definitional fluent.

In order to formalize regression, we first have to define which formulas we can handle:
Definition 3.27 (Regressable Formulas). Let \( \Sigma \) be a basic action theory over \( \langle D, F \rangle \). An objective formula \( \alpha \) is regressable iff it is bounded and the only fluents mentioned are from \( \langle D, F \rangle \).

The restriction to bounded formulas is because while we can easily treat \([t]\) operators as explained above, \(\Box\) operators are much more difficult and are therefore not handled by standard regression; we will discuss this issue later in this thesis in Chapter 5. Furthermore we of course need as a requirement that the formula to be regressed only mentions fluents for which we actually have the corresponding axioms in our basic action theory.

In order to be able to deal with functional fluents in a similar manner as we handle relational ones, we have the following definition:

Definition 3.28 (Regressable Form). A regressable formula \( \alpha \) is in regressable form if every quantifier uses a distinct variable and every occurrence of a functional fluent \( f \) in \( \alpha \) is of the form \( f(\vec{t}) = t' \), where \( \vec{t} \) and \( t' \) do not contain any further functional fluents.

Restricting ourselves to formulas of this form does not constitute a loss of generality:

Lemma 3.29. Every regressable formula \( \alpha \) can be expressed in regressable form.

Proof. Quantified variables can be made distinct by simply renaming them appropriately.

The requirement on functional fluents can be achieved by introducing additional existential quantifiers. For example, if \( f \) is a unary and \( g \) a zero-ary functional fluent, then \( F(f(g)) \) in regressable form is

\[ \exists x \exists y. g = x \land f(x) = y \land F(y). \]

Note that the regressable form of a formula is at most linear in the size of the original one. Using the above, we will in the following simply identify regressable formulas with formulas in regressable form. We are now ready to define the regression operator:

Definition 3.30 (The Regression Operator). We define \( R[\alpha] \), the regression of \( \alpha \) wrt a basic action theory \( \Sigma \), to be \( R[\langle \rangle, \alpha] \), where for any sequence of action terms \( \sigma \), \( R[\sigma, \alpha] \) is defined inductively on \( \alpha \) by:

1. \( R[\sigma, (t_1 = t_2)] = (t_1 = t_2); \)
2. \( R[\sigma, \alpha \land \beta] = R[\sigma, \alpha] \land R[\sigma, \beta]; \)
3. \( R[\sigma, \neg \alpha] = \neg R[\sigma, \alpha]; \)
4. \( R[\sigma, \forall x. \alpha] = \forall x. R[\sigma, \alpha] \) (for all sorts);
3.4 Projection

5. \( R[\sigma, [t] \alpha] = R[\sigma \cdot t, \alpha] \);

6. \( R[\sigma, G(I)] = G(I) \), where \( G \) is a rigid predicate;

7. \( R[\sigma, F(I)] \) for relational normal fluents \( F \) is defined inductively by:
   (a) \( R[\langle \rangle, F(I)] = F(I) \);
   (b) \( R[\sigma \cdot t, F(I)] = R[\sigma, \gamma F_{\vec{x}}] \);

8. \( R[\sigma, (f(I) = t')] \) for functional normal fluents \( f \) is defined inductively by:
   (a) \( R[\langle \rangle, (f(I) = t')] = (f(I) = t') \);
   (b) \( R[\sigma \cdot t, (f(I) = t')] = R[\sigma, \gamma f_{\vec{x}} y] \);

9. for relational definitional fluents \( D \):
   \( R[\sigma, D(I)] = R[\sigma, \varphi D_{\vec{x}}] \)

10. for functional definitional fluents \( d \):
    \( R[\sigma, (d(I) = t')] = R[\sigma, \varphi d_{\vec{x}} y] \)

To prove the regression operator’s correctness, and also for the definition of progression in the next subsection, we need the following definition:

**Definition 3.31.** Let \( w \) be a world and \( \Sigma \) a basic action theory over fluents \( \langle D, F \rangle \). Then \( w_{\Sigma} \) is a world satisfying the following conditions:

1. For any functional \( g \notin D \cup F \) and relational \( G \notin D \cup F \):
   \begin{align*}
   w_{\Sigma}[g(n_1, \ldots, n_k), z] &= w[g(n_1, \ldots, n_k), z]; \\
   w_{\Sigma}[G(n_1, \ldots, n_k), z] &= w[G(n_1, \ldots, n_k), z];
   \end{align*}

2. for any functional \( f \in F \):
   \begin{align*}
   w_{\Sigma}[f(n_1, \ldots, n_k), \langle \rangle] &= w[f(n_1, \ldots, n_k), \langle \rangle]; \\
   w_{\Sigma}[f(n_1, \ldots, n_k), z \cdot p] &= n \text{ iff } w_{\Sigma}, z \models \gamma f_{x_1 \ldots x_k y} a;
   \end{align*}

3. for any relational \( F \in F \):
   \begin{align*}
   w_{\Sigma}[F(n_1, \ldots, n_k), \langle \rangle] &= w[F(n_1, \ldots, n_k), \langle \rangle]; \\
   w_{\Sigma}[F(n_1, \ldots, n_k), z \cdot p] &= 1 \text{ iff } w_{\Sigma}, z \models \gamma F_{x_1 \ldots x_k a};
   \end{align*}

4. for any functional \( d \in D \) and relational \( D \in D \):
   \begin{align*}
   w_{\Sigma}[d(n_1, \ldots, n_k), z] &= n \text{ iff } w_{\Sigma}, z \models \varphi d_{a_1 \ldots a_k}; \\
   w_{\Sigma}[D(n_1, \ldots, n_k), z] &= 1 \text{ iff } w_{\Sigma}, z \models \varphi D_{a_1 \ldots a_k}.\]
The above definition as well as the following lemma and theorem are similar to the ones in [LL04] and [LL10], but tailored to our specific variant of $\mathcal{E}S$.

**Lemma 3.32.** Let $w$ be a world, $\Sigma = \Sigma_0 \cup \Sigma_{def} \cup \Sigma_{post}$ a basic action theory, and $\alpha$ a regressable formula. Then

1. $w_\Sigma$ exists and is uniquely defined.
2. If $w \models \Sigma_0$, then $w_\Sigma \models \Sigma$.
3. If $w \models \Sigma$ then $w = w_\Sigma$.
4. $w_\Sigma \models \alpha$ iff $w \models R[\alpha]$.
5. If $\alpha$ is a fluent sentence, then $R[\alpha] = \alpha$.
6. If $\alpha$ is a fluent sentence, then $w_\Sigma \models \alpha$ iff $w \models \alpha$.

**Proof.** (Sketch)

1. $w_\Sigma$ clearly exists. The uniqueness follows from the fact that the $\varphi_D$ and $\varphi_d$ are fluent formulas wrt $\langle D, F \rangle$ and that for all fluents in $F$, once their initial values are fixed, then the values after any number of actions are uniquely determined by $\Sigma_{post}$.

2. Directly from the definition of $w_\Sigma$ we get that $w_\Sigma \models \Box D(\vec{x}) \equiv \varphi_D$, $w_\Sigma \models \Box(d(\vec{x}) = y) \equiv \varphi_d$, $w_\Sigma \models \Box[a]F(\vec{x}) \equiv \gamma_F$, and $w_\Sigma \models \Box[a](f(\vec{x}) = y) \equiv \gamma_f$.

3. If $w_\Sigma \models \Box D(\vec{x}) \equiv \varphi_D$, $w_\Sigma \models \Box(d(\vec{x}) = y) \equiv \varphi_d$, $w_\Sigma \models \Box[a]F(\vec{x}) \equiv \gamma_F$, and $w_\Sigma \models \Box[a](f(\vec{x}) = y) \equiv \gamma_f$, then $w$ satisfies the definition of $w_\Sigma$.

4. For this property we prove a more general property. Assuming that $\sigma = \langle t_1, \ldots, t_k \rangle$,

$$w_\Sigma \models [t_1] \cdots [t_k] \alpha \text{ iff } w \models R[\sigma, \alpha].$$

(3.113)

The proof is simple, but tedious and we will skip the details here. The most interesting aspect about it is probably its structure, which is as follows. First, property (3.113) is proven for static formulas only by means of an induction over the length of $\sigma$ and a sub-induction over the length of $\alpha$, where we count the number of logical operators and occurrences of $D(\vec{t})$ and $(d(\vec{t}) = t')$ are counted as the length of $\varphi_D^{\vec{t}, 1}$ and $\varphi_d^{\vec{t}, y} + 1$, respectively. The induction is well-behaved because the formulas $\varphi_D$, $\varphi_d$, $\gamma_F$ and $\gamma_f$ are themselves fluent formulas wrt $\langle D, F \rangle$, that is they are static and do not mention any $D \in \mathcal{D}$ or $d \in \mathcal{D}$. 
After proving the property for static $\alpha$, the case for bounded formulas is treated by another simple induction on the number of $[t]$ operators in $\alpha$.

5. This property can be proven by a simple induction on the structure of $\alpha$. Since $\alpha$ is static, regression rules 5, 7b, and 8b never apply. Because $\alpha$ is fluent, it does also not contain any definitionals, hence rules 9 and 10 do not apply either. The remaining rules leave the input formula unchanged.

6. This property follows directly from properties (4) and (5).

**Theorem 3.33** (Regression Theorem). Let $\Sigma$ be a basic action theory and $\alpha$ a regresstable sentence. Then $R[\alpha]$ is a fluent sentence and satisfies

$$\Sigma \models \alpha \text{ iff } \Sigma_0 \models R[\alpha].$$

**Proof.** “$\Rightarrow$”: Suppose $\Sigma \models \alpha$. Let $w$ be a world such that $w \models \Sigma_0$. From Lemma 3.32 (2) it follows that $w_\Sigma \models \Sigma$, and hence $w_\Sigma \models \alpha$. By Lemma 3.32 (4) we obtain that $w \models R[\alpha]$.

“$\Leftarrow$”: Conversely, suppose $\Sigma_0 \models R[\alpha]$ and let $w$ be a world with $w \models \Sigma$. Then in particular $w \models \Sigma_0$, and so $w \models R[\alpha]$. By Lemma 3.32 (4), $w_\Sigma \models \alpha$. By Lemma 3.32 (3), $w_\Sigma = w$, and hence $w \models \alpha$.

**Example 3.34.** To illustrate how regression works we will apply it to verify the entailments (3.103) to (3.109) of our office robot’s basic action theory as presented in Example 3.26.

- $\Sigma \models \neg Holding(cup)$:
  We have $R[\langle \rangle, \neg Holding(cup)] = \neg R[\langle \rangle, Holding(cup)] = \neg Holding(cup)$. Since the initial database contains $\neg \exists x. Holding(x)$ (3.85), obviously $\Sigma_0 \models R[\neg Holding(cup)]$.

- $\Sigma \not\models Poss(pickup(cup))$:
  Using the precondition axiom (3.98), we obtain that $R[\langle \rangle, Poss(pickup(cup))]$ is

$$\{ \exists y (pickup(cup) = goto(y) \land Location(y) \land NextTo(loc(robot), y)) \lor$$
$$\exists y (pickup(cup) = pickup(y) \land loc(y) = loc(robot) \land \neg \exists x. Holding(x) \land \neg Heavy(y)) \lor$$
$$\exists y (pickup(cup) = putdown(y) \land Holding(y)) \lor$$
$$pickup(cup) = recharge \land \exists x (ChargingStation(x) \land loc(x) = loc(robot)) \}$$

$\land energy > 0$.

Due to unique names for actions, we can replace $(pickup(cup) = pickup(y))$ by $(cup = y)$ as well as all of $(pickup(cup) = goto(y)), (pickup(cup) = putdown(y))$ and $(pickup(cup) =$
recharge) by ⊥. Since three of the above disjuncts are thus unsatisfiable, we can simplify the regression result to

$$\text{loc}(\text{cup}) = \text{loc}(\text{robot}) \land \lnot \exists x \text{Holding}(x) \land \lnot \text{Heavy}(\text{cup}) \land \text{energy} > 0$$

which is not entailed by $\Sigma_0$ due to it containing the sentences $\text{loc}(\text{robot}) = \text{hallway}$ (3.87) and $\text{loc}(\text{cup}) = \text{kitchen}$ (3.88), and due to the fact that the distinct standard names hallway and kitchen are unequal.

- $\Sigma \models \text{Poss}(\text{goto}(\text{loc}(\text{cup})))$:
  Again using the precondition axiom and similar simplification as in the last item, we get that $R[\langle \rangle, \text{Poss}(\text{goto}(\text{loc}(\text{cup})))]$ is equivalent to

$$\text{Location}(\text{loc}(\text{cup})) \land \text{NextTo}(\text{loc}(\text{robot}), \text{loc}(\text{cup}))$$

which is entailed by $\Sigma_0$ due to $\text{loc}(\text{robot}) = \text{hallway}$ (3.87), $\text{loc}(\text{cup}) = \text{kitchen}$ (3.88), as well as $\text{NextTo}(\text{hallway}, \text{kitchen})$ (3.90),(3.91), and $\text{Location}(\text{kitchen})$ (3.96),(3.97).

- $\Sigma \models [\text{goto}(\text{loc}(\text{cup}))] \mid \text{loc}(\text{robot}) = \text{kitchen}$:
  If $t = \text{goto}(\text{loc}(\text{cup}))$, we have $R[\langle \rangle, [t] \text{loc}(\text{robot}) = \text{kitchen}] = R[\langle t \rangle, \text{loc}(\text{robot}) = \text{kitchen}]$, which yields with the successor state axiom (3.99) and unique action names

$$\text{loc}(\text{cup}) = \text{kitchen} \land (\text{robot} = \text{robot} \lor \text{Holding}(\text{robot})),$$

what can be further reduced to $\text{loc}(\text{cup}) = \text{kitchen}$. Hence, in order for the robot to be in the kitchen after moving to the location of the cup, it is necessary and sufficient that the cup is located in the kitchen. This is indeed the case according to the robot’s initial database (3.88).

- $\Sigma \models [\text{goto}(\text{loc}(\text{cup}))] \mid \text{energy} = 38$:
  In order to determine $R[\langle \text{goto}(\text{loc}(\text{cup})) \rangle, \text{energy} = 38]$ we use the successor state axiom for energy (3.101):

$$38 = \text{energy} - 3 \cdot \text{distance}(\text{loc}(\text{cup}), \text{loc}(\text{robot}))$$

The equation is entailed by $\Sigma_0$ due to $\text{energy} = 50$ (3.86) and since

$$\text{distance}(\text{loc}(\text{cup}), \text{loc}(\text{robot})) = \text{distance}(\text{kitchen}, \text{hallway}) = 4$$

by (3.92) and (3.93).
• \( \Sigma \models [\text{goto}(\text{loc}(\text{cup})))\text{Poss}(\text{pickup}(\text{cup}))]: \)

Let \( t = \text{pickup}(\text{cup}) \). Similar to above, we obtain that \( R[[t], \text{Poss}(\text{pickup}(\text{cup}))] \) is

\[
R[[t], \text{loc}(\text{cup})] = \text{loc}(\text{robot}) \land \neg \exists x \text{Holding}(x) \land \neg \text{Heavy}(\text{cup}) \land \text{energy} > 0
\]

which according to our convention regressing functional fluents we can rewrite as

\[
R[[t], \exists y. \text{loc}(\text{cup}) = y \land \text{loc}(\text{robot}) = y \land \neg \exists x \text{Holding}(x) \land \neg \text{Heavy}(\text{cup})
\]

\[\land \exists y'. \text{energy} = y' \land y' > 0] \]

Regressing \( \text{loc}(\text{cup}) = y \) through \( t \) has no effect: the resulting formula can be reduced to \( \text{loc}(\text{cup}) = y \) again. The regression of \( \text{loc}(\text{robot}) = y \) through \( t \) moreover is \( \text{loc}(\text{cup}) = y \) as well: The robot’s location after moving to the location of the cup is of course the previous location of the cup. \( \text{Holding} \) and \( \text{Heavy} \) are again unaffected by regression through \( t \) since the \( \text{goto} \) action does not change the value of the former, and the latter is a rigid predicate. Finally, \( \text{energy} = y' \) is regressed similarly to as we did in the last item, only using \( y' \) as the value instead of 38. Putting it all together, we have:

\[
\exists y. \text{loc}(\text{cup}) = y \land \neg \exists x \text{Holding}(x) \land \neg \text{Heavy}(\text{cup})
\]

\[\land \exists y'. \text{energy} = y' \land y' > 0] \]

The existence of a location of the cup is a tautology, the robot is initially not holding anything (3.85), the cup is not heavy (3.89), the energy level is 50 (3.86), and the distance between the cup’s location (the kitchen) and the robot’s (the hallway) is 4, as before. Therefore, the inequation holds and the regression result is entailed by the initial theory.

• \( \Sigma \models [\text{goto}(\text{loc}(\text{cup})))\text{[pickup}(\text{cup})]\text{Holding}(\text{cup})]: \)

We need to determine \( R[[\text{goto}(\text{loc}(\text{cup}))), \text{pickup}(\text{cup})], \text{Holding}(\text{cup})] \). Note that regressing \( \text{Holding}(\text{cup}) \) through \( \text{pickup}(\text{cup}) \) immediately simplifies to \( \top \), whose regression through \( \text{goto}(\text{loc}(\text{cup})) \) is of course again \( \top \). Therefore obviously, the result is entailed.

### 3.4.2 Progression

While regression is an elegant solution to the projection problem, it also has its drawbacks. For one thing, note that regressing a formula tends to blow up its size exponentially. Intuitively, this is because every fluent atom \( F(\vec{t}) \) or \( f(\vec{t}) = t' \) within the scope of a \([\cdot]\) operator is replaced by a formula that consists of at least one, but in general multiple such atoms, say \( K \) on average. When the formula contains \( N \) nested \([\cdot]\) operators (i.e. talks about action sequences up to length \( N \)), then the regression result will contain around \( K^N \) fluent atoms. In consequence, the
regressed formulas soon become unmanageable in size. This gets worse with every new action the robot performs, and a robot that has been in operation for some time will have acquired a very long history of already executed actions.

Apart from the problem of the sheer size of regression results, another point is that typically, the entire history will contain a lot of redundancies, and only the most recent history is really relevant for a certain task. In order for the robot to bring the cup of coffee to Bob, it is not so important where the cup has been put and moved around during the last two years, but when it is known that the cup was stored in the kitchen yesterday and has not been touched since then, this is actually all that matters.

Finally, regression also needs to be done anew for every single query formula. While it is conceivable to come up with a caching mechanism to cope with the redundancies resulting from a repeated evaluation of formulas and subformulas, the idea behind progression is instead to update the knowledge base $\Sigma_0$ such that it does not describe the initial situation any longer, but reflects the state of the world at the present situation. Thus, the agent “forgets” its history of already performed actions and only keeps that information which is of relevance for the present and future. As opposed to regression, the progression of the knowledge base only needs to be computed once for the action sequence in question, and every corresponding query can then be directly checked against this new knowledge base.

Furthermore, in particular a physical agent like a robot often has phases of “mental idle time” during which it is busy performing physical actions (like moving from one room to the next), but where it does not have to perform any reasoning tasks (like deciding on the next action). Such time periods can ideally be used to progress the knowledge base through the history of the latest actions that have been done, and thus bring the robot’s internal representation of the state of the outside world up to date.

Before we can apply progression on knowledge bases, we first have to provide a semantic definition of what constitutes a progression. We start with defining progression for single worlds:

**Definition 3.35** (Progression of a World). Let $w$ be a world and $z$ a sequence of action standard names. The *progression of $w$ through $z$*, denoted as $w_z$, is a world such that

- $w_z[\beta, z'] = w[\beta, z \cdot z']$ for all primitive formulas $\beta$ and all $z' \in Z$ and
- $w_z[t, z'] = w[t, z \cdot z']$ for all primitive terms $t$ and all $z' \in Z$.

In other words, $w_z$ is the world that we obtain when we “cut off” the subtree starting at $z$ in $w$. The following lemma shows how the definition generalizes to arbitrary ground terms and formulas:
Lemma 3.36. Let $w$ be a world, $z, z'$, sequences of action standard names, $\alpha$ an objective SO formula, and $t$ a ground term. Then

1. $|t|^{z'}_{w,z} = |t|^z_{w} z'$

2. $w, z \models \alpha$ iff $w, z \cdot z' \models \alpha$

Proof.

1. The proof is by induction on the structure of $t$.

   - If $t \in \mathcal{N}$, then $|t|^{z'}_{w,z} = t = |t|^z_{w} z'$.
   - If $t = f(t_1, \ldots, t_k)$:
     
     
     \[
     |f(t_1, \ldots, t_k)|^{z'}_{w,z} = w_z[f(n_1, \ldots, n_k), z'], \text{ where } n_i = |t_i|^{z'}_{w,z} \quad \text{(by the semantics)}
     \]
     
     
     \[
     = w_z[f(n_1, \ldots, n_k), z'], \text{ where } n_i = |t_i|^z_{w} z' \quad \text{(by induction)}
     \]
     
     
     \[
     = w[f(n_1, \ldots, n_k), z \cdot z'], \text{ where } n_i = |t_i|^{z'}_{w} \quad \text{(by Definition 3.35)}
     \]
     
     
     \[
     = |f(t_1, \ldots, t_k)|^{z \cdot z'}_{w} \quad \text{(by the semantics)}
     \]

2. The proof is by induction on the structure of $\alpha$.

   - $\alpha = F(\bar{t})$:
     
     
     \[
     w_z, z' \models \alpha = \alpha = F(t_1, \ldots, t_k)
     \]
     
     
     \[
     \text{iff } w_z[F(n_1, \ldots, n_k), z'] = 1, \text{ where } n_i = |t_i|^{z'}_{w,z} \quad \text{(by the semantics)}
     \]
     
     
     \[
     \text{iff } w_z[F(n_1, \ldots, n_k), z'] = 1, \text{ where } n_i = |t_i|^z_{w} z' \quad \text{(by item 1 of this lemma)}
     \]
     
     
     \[
     \text{iff } w[F(n_1, \ldots, n_k), z \cdot z'] = 1, \text{ where } n_i = |t_i|^{z'}_{w} \quad \text{(by Definition 3.35)}
     \]
     
     
     \[
     \text{iff } w, z \cdot z' \models F(t_1, \ldots, t_k) \quad \text{(by the semantics)}
     \]

   - $\alpha = (t_1 = t_2)$:
     
     
     \[
     w_z, z' \models (t_1 = t_2)
     \]
     
     
     \[
     \text{iff } |t_1|^{z'}_{w,z} = |t_2|^{z'}_{w,z} \quad \text{(by the semantics)}
     \]
     
     
     \[
     \text{iff } |t_1|^z_{w} z' = |t_2|^z_{w} z' \quad \text{(by item 1 of this lemma)}
     \]
     
     
     \[
     \text{iff } w, z \cdot z' \models (t_1 = t_2) \quad \text{(by the semantics)}
     \]
• $\alpha = \alpha_1 \land \alpha_2$:
  
  \[ w_z, z' \models \alpha_1 \land \alpha_2 \]
  
  iff $w_z, z' \models \alpha_1$ and $w_z, z' \models \alpha_2$ (by the semantics)
  
  iff $w, z \cdot z' \models \alpha_1$ and $w, z \cdot z' \models \alpha_2$ (by induction)
  
  iff $w, z \cdot z' \models \alpha_1 \land \alpha_2$ (by the semantics)

• $\alpha = \neg \alpha_1$:
  
  \[ w_z, z' \models \neg \alpha_1 \]
  
  iff $w_z, z' \not\models \alpha_1$ (by the semantics)
  
  iff $w, z \cdot z' \not\models \alpha_1$ (by induction)
  
  iff $w, z \cdot z' \models \neg \alpha_1$ (by the semantics)

• $\alpha = \forall x \alpha_1$:
  
  \[ w_z, z' \models \forall x \alpha_1 \]
  
  iff $w_z, z' \models \alpha_1^n$ for all $n \in \mathbb{N}_x$ (by the semantics)
  
  iff $w, z \cdot z' \models \alpha_1^n$ for all $n \in \mathbb{N}_x$ (by induction)
  
  iff $w, z \cdot z' \models \forall x \alpha_1$ (by the semantics)

• $\alpha = [t] \alpha_1$:
  
  \[ w_z, z' \models [t] \alpha_1 \]
  
  iff $w_z, z' \cdot p \models \alpha_1$, where $p = |t|_{w_z} z'$ (by the semantics)
  
  iff $w_z, z' \cdot p \models \alpha_1$, where $p = |t|_{w_z} z'$ (by item 1 of this lemma)
  
  iff $w, z \cdot z' \cdot p \models \alpha_1$, where $p = |t|_{w} z'$ (by induction)
  
  iff $w, z \cdot z' \models [t] \alpha_1$ (by the semantics)

• $\alpha = \Box \alpha_1$:
  
  \[ w_z, z' \models \Box \alpha_1 \]
  
  iff $w_z, z' \cdot z'' \models \alpha_1$ for all $z'' \in Z$ (by the semantics)
  
  iff $w, z \cdot z' \cdot z'' \models \alpha_1$ for all $z'' \in Z$ (by induction)
  
  iff $w, z \cdot z' \models \Box \alpha_1$ (by the semantics)

• $\alpha = \forall P \alpha_1$:
  
  \[ w_z, z' \models \forall P \alpha_1 \]
  
  iff $w', z' \models \alpha_1$ for all $w' \sim P w_z$ (by the semantics)
iff \( w''_z, z' \models \alpha_1 \) for all \( w''_z \sim_P w_z \) (see below)

iff \( w'', z \cdot z' \models \alpha_1 \) for all \( w''_z \sim_P w_z \) (by induction)

iff \( w'', z \cdot z' \models \alpha_1 \) for all \( w'' \sim_P w \) (see below)

iff \( w, z \cdot z' \models \forall P \alpha_1 \) (by the semantics)

In the second rewriting step above, we exploited the fact that every world \( w' \) is expressible as \( w''_z \) for some \( w'' \). The fourth step makes use of the fact that \( w''_z \sim_P w_z \)
iff \( w' \sim_P w \), which is a direct consequence of Definitions 3.38 and 3.35.

- \( \alpha = \forall f \alpha_1 \):

The proof of this case is exactly like the one for \( \forall P \alpha_1 \) above.

We can now define progression for knowledge bases:

**Definition 3.37** (Progression). Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory. A set of static, objective SO sentences \( \Sigma' \) is a **progression** of \( \Sigma \) through a fluent action term \( t \) wrt \( \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) iff for any world \( w' \), \( w' \models \Sigma' \) iff there is a world \( w \) such that \( w \models \Sigma \) and \( w'_{\Sigma} = w_p \), where \( p = |t|_w \).

Intuitively, the definition means that a set of formulas \( \Sigma' \) is a progression when for all and only its models \( w' \), a model \( w \) of the original theory can be reconstructed. An observer who is only looking “forward” in time would thus be unable to distinguish a model \( w' \) of \( \Sigma' \) viewed from the initial situation from a model \( w \) of the original theory \( \Sigma \) viewed from the situation after performing action \( t \). The following theorem shows that this definition of progression is correct in the sense that \( \Sigma' \) is the strongest post-condition of \( \Sigma \) (cf. e.g. [Ped86]):

**Theorem 3.38.** Let \( \Sigma' \) be a progression of \( \Sigma_0 \) through \( t \) wrt \( \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). Then for any static, objective SO sentence \( \phi \),

\[
\Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models [t] \phi \iff \Sigma'_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \phi.
\]

**Proof.** “\( \Rightarrow \)”: Let \( \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models [t] \phi \) and \( w' \) be a world such that \( w' \models \Sigma'_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). We have to show that \( w' \models \phi \). First note that by Lemma 3.32 (3) we have \( w' = w'_{\Sigma} \). By Definition 3.37, there exists a world \( w \) such \( w \models \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), where \( w_p = w' \) and \( p = |t|_w \). By assumption, \( w \models [t] \phi \), i.e. \( w, p \models \phi \). Then by Lemma 3.36 (2), we get \( w_p, () \models \phi \), i.e. \( u' \models \phi \).

“\( \Leftarrow \)”: Let \( \Sigma'_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \phi \), let \( w \) be a world with \( w \models \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), let \( u' = w_p \) and \( p = |t|_w \). By Lemma 3.32 (3), \( w = w_{\Sigma} \), and hence also \( u' = w'_{\Sigma} \). By Definition 3.37, \( u' \models \Sigma'_0 \), and therefore \( w' \models \Sigma'_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) by Lemma 3.32 (2). Then by assumption, \( u' \models \phi \), and by Lemma 3.36 (2) we get \( w, p \models \phi \), i.e. \( w \models [t] \phi \).
Now that we defined what constitutes a progression, the next question is how we determine
the progression of a given knowledge base. In [LR97], Lin and Reiter show that as long as we
can resort to second-order quantification, a progression is guaranteed to exist and can easily be constructed:

**Theorem 3.39 (LR-Progression).** Let \( \Sigma = \Sigma_0 \cup \Sigma_{def} \cup \Sigma_{post} \) be a basic action theory over fluents \( \langle D, F \rangle \). For every relational \( F_i \in F \), we introduce a new predicate symbol \( P_i \) with the same
arity. Similarly, for every functional \( f_i \in F \), let \( h_i \) be a new function symbol of the same arity
and sort. For any formula \( \phi \), let \( \phi_{P, h}^{F, f} \) denote the result of replacing every occurrence of \( F_i \) by
\( P_i \) and every occurrence of \( f_i \) by \( h_i \). Then the following is a progression of \( \Sigma_0 \) through \( t \) wrt \( \Sigma_{post} \):

\[
(3.114) \quad \exists P_1 \ldots \exists P_l \exists h_1 \ldots \exists h_m. \Psi,
\]

where \( \Psi \) refers to the conjunction of the following:

\[
(3.115) \quad (\Sigma_0)_{P, h}^{F, f}
\]

\[
(3.116) \quad \bigwedge_{i=1}^{l} \forall \bar{x}_i (F_i(\bar{x}_i) \equiv (\gamma_{F_i})_{P, h}^{F, f})
\]

\[
(3.117) \quad \bigwedge_{i=1}^{m} \forall \bar{x}_i \forall y_i ((f_i(\bar{x}_i) = y_i) \equiv (\gamma_{f_i})_{P, h}^{F, f})
\]

**Proof.** Here we only provide the structure of the proof. The complete proof can be found in
Appendix A.1. Let \( w' \) be a world and let \( \Sigma_0' \) denote (3.114).

\( \Rightarrow \): Let \( w' \models \Sigma_0' \). We have to show that there is some world \( w \) such that \( w \models \Sigma \) and
\( w^t = w_p, \) where \( p = |t|_w^0 \). By Definition 3.22, there is a world \( w'' \) such that \( w'' \models \psi \) and
\( w'' \models \Psi. \) We define \( w''' \) to be a world such that for all the \( F_i, \) all the \( f_i \) and all standard names \( \bar{n}, \)

\[
w'''[F_i(\bar{n}), \langle \rangle] = w''[P_i(\bar{n}), \langle \rangle]
\]

\[
w'''[f_i(\bar{n}), \langle \rangle] = w''[h_i(\bar{n}), \langle \rangle]
\]

and for all \( G, g \notin D \cup F \cup \{P_1, \ldots, P_l\} \cup \{h_1, \ldots, h_m\}, \) all \( \bar{n}, \) and all \( z \in Z, \)

\[
w'''[G(\bar{n}), p \cdot z'] = w''[G(\bar{n}), z']
\]

\[
w'''[g(\bar{n}), p \cdot z'] = w''[g(\bar{n}), z']
\]

and further that for all \( P_i, \) all \( h_i, \) all \( \bar{n}, \) and all \( z' \in Z, \)

\[
w'''[P_i(\bar{n}), p \cdot z'] = w'[P_i(\bar{n}), z']
\]

\[
w'''[h_i(\bar{n}), p \cdot z'] = w'[h_i(\bar{n}), z']
\]
Now let $w = w_{\Sigma}^m$. We will need the property that for any fluent sentence $\phi$,

\[(3.118) \quad w'' \models \phi_{F_f P_h}^I \iff w''' \models \phi,\]

which can be shown by an induction on the structure of $\phi$. Since $w'' \models (\Sigma_0)_{F_f P_h}^I$ by assumption, it now follows from (3.118) that $w''' \models \Sigma_0$. By Lemma 3.32 (2) we get $w_{\Sigma}^m \models \Sigma$.

As for showing that $w_{\Sigma}^\prime = w_p$, according to Definition 3.35, this means that for all $z' \in Z$, all primitive formulas $\beta$ and all primitive terms $t'$:

\[w_{\Sigma}^\prime[t',z'] = w[t',p \cdot z'] \quad (3.119)\]
\[w_{\Sigma}^\prime[\beta,z'] = w[\beta,p \cdot z'] \quad (3.120)\]

To prove this, we show a more general property, namely that for all ground terms $t''$ and all static, objective sentences $\phi$:

\[|t''|_{w_{\Sigma}^\prime}[z'] = |t''|_{w}[z'] \quad (3.121)\]
\[w_{\Sigma}^\prime[z',z'] \models \phi \iff w_{\Sigma}^\prime[p \cdot z'] \models \phi \quad (3.122)\]

Properties (3.119) and (3.120) follow then as special cases from (3.121) and (3.122), respectively, since any primitive term is also a ground term, and any primitive formula is also a static, objective sentence. The proof is on both theses properties together, with an outer induction on the length of the action sequence $z'$ and a sub-induction on the size of $t''$ and $\phi$, where any occurrence of $d(t)$ is counted as the size of the corresponding $\varphi_d^I + 1$ and any occurrence of $D(t)$ as the size of $\varphi_D^I + 1$. The induction is well-behaved since the formulas $\varphi_d, \varphi_D, \gamma_f$ and $\gamma_F$ are fluent formulas wrt $\langle F, D \rangle$ and therefore do not mention any further $d \in D$ or $D \in D$.

"$\Rightarrow$": Let $w$ be a world such that $w \models \Sigma$ and $w_p = w_{\Sigma}^\prime$, where $p = |t|_w$. We have to show that $w' \models \Sigma_0$, or equivalently that there exists some $w''$ with $w'' \sim_{\bar{F}_h} w'$ such that $w'' \models \Psi$. Let $w''$ be a world that is like $w'$, except that for every $P_i$, every $f_i$, and all names $\vec{n}$,

\[w''[P_i(\vec{n}),()] = w[F_i(\vec{n}),()],\]
\[w''[f_i(\vec{n}),()] = w[f_i(\vec{n}),()].\]

Obviously $w'' \sim_{\bar{F}_h} w'$. In order to show that $w'' \models \Psi$, we first need the property that for every fluent formula $\phi$,

\[(3.123) \quad w \models \phi \iff w'' \models \phi_{F_f P_h}^I,\]

which is again provable by means of an induction on $\phi$’s structure. We can then show that $w \models \Psi$ as follows:
• \( w'' \models (\Sigma_0)^{\vec{F}}_{\vec{P} \vec{H}} \):
  This follows directly from the fact that \( w \models \Sigma_0 \) and (3.123).

• \( w'' \models \forall \vec{x}_i (F_i(\vec{x}) \equiv (\gamma F_i^{\vec{a}})^{\vec{F}}_{\vec{P} \vec{H}}) \):
  Let \( \vec{n} \) be arbitrary standard name instances for the \( \vec{x}_i \). We have that

\[
\begin{align*}
  w'' &\models F_i(\vec{n}) \\
  &\iff w''[F_i(\vec{n}), \langle \rangle] = 1 \quad \text{(by the semantics)} \\
  &\iff w''[F_i(\vec{n}), \langle \rangle] = 1 \quad \text{(by assumption)} \\
  &\iff w'_i[F_i(\vec{n}), \langle \rangle] = 1 \quad \text{(by Definition 3.31)} \\
  &\iff w_p[F_i(\vec{n}), \langle \rangle] = 1 \quad \text{(since } w_p = w'_i \text{ by assumption)} \\
  &\iff w_p[F_i(\vec{n}), p] = 1 \quad \text{(by Definition 3.35)} \\
  &\iff w, p \models F_i(\vec{n}) \quad \text{(by the semantics)} \\
  &\iff w \models \gamma F_i^{\vec{x}_i}_{\vec{P} \vec{H}} \quad \text{(since } w \models \Sigma_{\text{post}} \text{ by assumption)} \\
  &\iff w \models \gamma F_i^{\vec{x}_i}_{\vec{P} \vec{H}} \quad \text{(since } p = |t|_w \text{ by assumption)} \\
  &\iff w'' = (\gamma F_i^{\vec{x}_i}_{\vec{P} \vec{H}})^{\vec{F} \vec{P} \vec{H}} \quad \text{(by (3.123))} \\
  &\iff w'' = ((\gamma F_i^{\vec{a}})^{\vec{F} \vec{P} \vec{H}})^{\vec{x}_i} \quad \text{(see below)}
\end{align*}
\]

In the last rewriting step we made use of the fact that the \( \vec{x}_i \) and \( \vec{F} \vec{P} \vec{H} \) substitutions do not interfere with each other since the former is only concerned with variables and the latter only with predicate and function symbols, which is why their order can be reversed.

• \( w'' \models \forall \vec{x}_i \forall y_i (f_i(\vec{x}_i) = y_i \equiv (\gamma f_i^{\vec{a}})^{\vec{F} \vec{P} \vec{H}}) \):
  The proof in this case is very similar to the one above. \( \square \)

**Example 3.40.** Consider our office robot again. Suppose its initial database is

\[
\Sigma_0 = \{ \neg \exists x \text{Holding}(x) \land \text{loc(robot)} = \text{kitchen} \land \text{loc(cup)} = \text{kitchen} \}
\]

and that, for simplicity, \( \text{Holding} \) and \( \text{loc} \) are the only normal fluents. Assume that we want to progress through the action \( t = \text{pickup(cup)} \). First, let \( P \) be a new unary predicate symbol and \( h \) a new unary function symbol. Then

\[
(\Sigma_0)^{\text{Holding} \text{loc}}_P \text{ h} = \{ \neg \exists x P(x) \land h(\text{robot}) = \text{kitchen} \land \text{loc(cup)} = \text{kitchen} \}.
\]
Instantiating the successor state axioms for \textit{Holding} (3.100) and \textit{loc} (3.99) and substituting $P$ and $h$ on their right-hand side yields

\[ \forall x. \text{Holding}(x) \equiv \text{pickup}(\text{cup}) = \text{pickup}(x) \lor \neg P(x) \land \text{pickup}(\text{cup}) \neq \text{putdown}(x) \]

\[ \forall x \forall y. \text{loc}(x) = y \equiv \text{pickup}(\text{cup}) = \text{goto}(y) \land (x = \text{robot} \lor P(x)) \lor h(x) = y \land \neg \exists y'. \text{pickup}(\text{cup}) = \text{goto}(y') \land (x = \text{robot} \lor P(x)) \]

which we can simplify by unique action names to

\[ \forall x. \text{Holding}(x) \equiv x = \text{cup} \lor \neg P(x) \]

\[ \forall x \forall y. \text{loc}(x) = y \equiv h(x) = y. \]

The progression is then given by

\[ \exists P \exists h. \neg \exists x (P(x) \land \text{loc}(\text{robot}) = \text{kitchen} \land \text{loc}(\text{cup}) = \text{kitchen}) \land \forall x (H(x) \equiv x = \text{cup} \lor \neg P(x)) \land \forall x \forall y (\text{loc}(x) = y \equiv h(x) = y). \]

Obviously, a function $h$ that is equivalent to \textit{loc} exists. Moreover, we can pull subformulas out of the scope of quantifiers that do not mention any of the quantified symbols. Therefore, the above can be simplified to

\[ \text{loc}(\text{robot}) = \text{kitchen} \land \text{loc}(\text{cup}) = \text{kitchen} \land \exists P. \neg \exists x P(x) \land \forall x (H(x) \equiv x = \text{cup} \lor \neg P(x)). \]

Similarly, the above expresses that $P$ is false for all individuals $x$, hence $\neg P(x)$ always holds. We can therefore simplify once more to obtain

\[ \text{loc}(\text{robot}) = \text{kitchen} \land \text{loc}(\text{cup}) = \text{kitchen} \land \forall x (H(x) \equiv x = \text{cup}) \]

which is precisely what to expect when progressing through $\text{pickup}(\text{cup})$: the locations of the robot and the cup remain the same, but the set of instances of $\text{Holding}(x)$ changes from the empty set to the singleton \{\text{cup}\}.

Unfortunately, this form of second-order progression is problematic for several reasons. First, we argued above that it is sensible to assume that our agent is a first-, but not a second-order reasoner. Of course, such an agent would be unable to deal with a knowledge base that contains second-order quantifiers. Second, note that this form of progression amounts to “cheating”. Recall that the idea behind progression was to forget about the previous situation and only keep the information that is relevant to the current one. While syntactically, the LR-progression certainly has the right form in the sense that it treats the situation resulting
from doing action $t$ as the new initial one, actually no information about the original situation is dropped! Instead, by means of renaming through second-order quantification, the contents of the original knowledge base are kept in their entirety, and their connection to the current situation is encoded in an equally “hidden” instantiation of the successor state axioms. It cannot be expected that reasoning with this knowledge base is any easier than reasoning with the original one, e.g. using regression.

Sadly, Lin and Reiter’s landmark article on progression [LR97] further comes with a strong negative result: In general, it is not possible to represent the progression of a knowledge base solely through a finite collection of first-order formulas. Furthermore, Vassos and Levesque [VL08] showed that even the infinite set of all first-order entailments of a basic action theory about the situation resulting from executing the action $t$ is weaker than a progression through $t$. It is very likely that these results hold similarly for $\mathcal{ES}$, but non-trivial to adapt their corresponding proofs. The reason is that in both cases, the existence of so-called unnamed individuals is essential, where in a semantic model, an individual is unnamed whenever no ground term exists that refers to it. While it is straightforward to construct such structures in classical Tarskian semantics, no individual in any $\mathcal{ES}$ world is unnamed due to the existence of standard names. Since both proofs are quite involved, reproving Lin and Reiter’s and Vassos and Levesque’s results within $\mathcal{ES}$ is beyond the scope of this thesis. Because we do not expect to gain any new insights from them, we leave it at that.

Far more important for our purposes is the fact that it is possible to identify subclasses of basic action theories for which first-order progressions indeed can be guaranteed to exist. The idea is to syntactically restrict the types of formulas allowed within the initial database and the successor state axioms in a way such that the progression is first-order representable. Ideally, such a class of theories is furthermore closed under progression, i.e. the result of progression falls again into the same class such that progression can be iterated. Already in [LR97], Lin and Reiter provide a number of classes of such first-order progressable theories, which are the so-called relatively complete databases, the class of context-free action theories, as well as the class of BATs that correspond to STRIPS descriptions. All of them can be adapted to $\mathcal{ES}$ in a straightforward manner, and we will not go into the details here about the former two, but refer the interested reader to [Cla05]. The STRIPS case is the most interesting one for the topic of this thesis. In fact, it constitutes the basis for our results on the integration of planning into the Golog programming language to be presented in Chapter 4, so it is also there where we further elaborate on this.
3.5 Knowledge and Sensing

So far we have only been concerned with objective formulas, that is formulas that express facts about the state of the world and how the agent’s actions affect it. However, it is often useful if not necessary to be able to talk about the agent’s knowledge about its environment, and also in particular its lack thereof.

As an example, consider again example formula (3.2) from this chapter’s introduction, repeated below:

\[ \exists x. \text{In}(x, \text{office}(\text{bob})) \land \text{Person}(x) \land (x \neq \text{bob}) \] (3.2)

It expresses that there is some person other than Bob in Bob’s office, and we might say that this is a case where the robot does not know who the person in Bob’s office is. Of course this is only true as long as the robot does not have any further knowledge. If on the other hand the agent’s knowledge base also contains the formula

\[ \forall x. \text{In}(x, \text{office}(\text{bob})) \supset x = \text{ann} \] (3.124)

saying that Ann is the only person in Bob’s office, then obviously the agent again does know all the persons in Bob’s office. The problem now is that what the agent knows or does not know is not expressible within the logical formalism itself, at least in the form as we have used it so far.

While in some simple contexts it is perfectly sufficient to only have a meta-theoretic notion of the agent’s state of knowledge, it is in general much more convenient to extend the language by an epistemic modal operator $\text{Know}$. This means that, in addition to the current rules for the construction of formulas, we also say that $\text{Know}(\alpha)$ is well-formed formula whenever $\alpha$ is. For instance, the fact that the robot knows that Bob is in his office or in the kitchen is expressed as follows:

\[ \text{Know}(\text{In}(\text{bob}, \text{office}(\text{bob})) \lor \text{In}(\text{bob}, \text{kitchen})) \] (3.125)

To express an agent’s lack of knowledge we simply put a negation in front of the $\text{Know}$ operator:

\[ \neg \text{Know}(\text{In}(\text{bob}, \text{office}(\text{bob}))) \] (3.126)

The above expresses that $\text{In}(\text{bob}, \text{office}(\text{bob}))$ is not known to be true, which is different from knowing the expression to be false. The agent does not know whether Bob is in his office when both $\text{In}(\text{bob}, \text{office}(\text{bob}))$ and its negation are not believed:

\[ \neg \text{Know}(\text{In}(\text{bob}, \text{office}(\text{bob}))) \land \neg \text{Know}(\neg \text{In}(\text{bob}, \text{office}(\text{bob}))) \] (3.127)
The new modal operator now allows us to explicitly express both de dicto and de re knowledge within the logic. The former refers to the situation where we know that a certain individual exists, for example a person in Bob’s office:

\[(3.128)\quad \text{Know}(\exists x. \text{In}(x, \text{office}(\text{bob})) \land \text{Person}(x) \land (x \neq \text{bob}))\]

The latter expresses that is is even known who that individual is:

\[(3.129)\quad \exists x. \text{Know}(\text{In}(x, \text{office}(\text{bob})) \land \text{Person}(x) \land (x \neq \text{bob}))\]

Note that knowing a person in Bob’s office de re implies knowing de dicto, but not vice versa.

In addition to \textit{Know}, we will require one further modal operator. The reason is that it is otherwise not possible to express all that is known to the agent. Consider the sentence \textit{Know}(\text{In}(\text{bob}, \text{office}(\text{bob})))\): It says that the agent knows that Bob is in his office, but it does not entail whether the agent does or does not know the current location of Ann. \textit{Know}(\alpha) thus really only expresses that “at least \alpha is known”, but maybe even more. For this reason, we will use \textit{OKnow}(\alpha) to denote only-knowing \cite{Lev90} a formula \alpha, which means knowing \alpha, but nothing more and nothing less than it. Typically, an agent has some finite, objective knowledge base \Sigma, and we will be interested in determining whether \textit{OKnow}(\Sigma) \models \alpha, where \alpha talks about the agent’s knowledge in the current situation or after the execution of actions.

### 3.5.1 Formal Definition

Formally, we extend the set of objective formulas as follows:

**Definition 3.41 (Formulas).** Formulas are formed according to the set of rules obtained by extending the Definitions 3.2 and 3.17 by:

5. If \alpha is a formula, then so are \textit{Know}(\alpha) and \textit{OKnow}(\alpha).

Formulas where no predicate or function symbol and no \[\Box\] and \[t\] operators occur outside the scope of a \textit{Know} or \textit{OKnow} are called \textit{subjective}, and formulas without \textit{OKnow} basic.

Semantically, knowledge will be interpreted with a variant of the classical possible-world semantics \cite{Hin62, Kri63}. The idea is that a semantic model will now consist of two parts: On the one hand, a single world \(w \in W\), as before, representing the “real world”. Additionally, we have a set of worlds \(e \subseteq W\), called an \textit{epistemic state}. Intuitively, the agent considers each world \(w' \in e\) possible. Some formula \(\alpha\) then is known when every world in the epistemic state satisfies it, and \(\alpha\) is only known in \(e\) if \(e\) consists exactly of those worlds where \(\alpha\) is true. The more worlds are in \(e\), the less is known to the agent, and vice versa. In the extreme case, \(e\)
consists of all worlds, then the agent knows nothing, or only of a single world, then has perfect knowledge. We even allow that \( e = \emptyset \), which can happen when the agent has an inconsistent knowledge base. Also note that it is not a requirement that the actual world \( w \) is part of the epistemic state, i.e. the agent’s beliefs about what the real world is like may be wrong.

We also want the agent to be able to gather new information at runtime through sensing. As opposed to Lakemeyer and Levesque’s [LL09a] formalization, on which the following is based, we here let the agent not only sense truth values, but allow for the more general case that a sensing result is given by an object standard name. Sensing truth values can then still be emulated by having a corresponding action return one of the two distinguished standard names \( n_{\top} \) and \( n_{\bot} \). For simplicity, we moreover do not distinguish between physical and sensing actions. Instead we assume that any action returns some kind of sensing result, and be it only a default value such as “\( \text{ok} \)”. A special definitional fluent \( sf(p) \) is used to encode possible sensing results by including a corresponding axiom

\[
\Box sf(a) = y \equiv \phi_{sf}
\]

into the basic action theory, where \( \phi_{sf} \) is a fluent formula whose free variables are \( a \) and \( y \). We then need:

**Definition 3.42 (Sensing Compatibility).** Let \( w \) and \( w' \) be two worlds. We define \( w' \simeq_z w \) (read: \( w' \) and \( w \) agree on sensing results throughout \( z \)) inductively as follows:

1. \( w' \simeq_{\emptyset} w \) iff \( w' \) and \( w \) agree on all primitive terms of sort \( \text{action} \);
2. \( w' \simeq_{z \cdot p} w \) iff \( w' \simeq_z w \) and \( w'[sf(p), z] = w[sf(p)] \).

Above, the additional condition that the worlds agree on the primitive terms of sort \( \text{action} \) is necessary in order to be sure that an action term such as \( \text{goto(kitchen)} \) is mapped to the same standard name \( p \) by \( w \) and \( w' \).

Similar to worlds, we now have to progress epistemic states as well. This amounts to progressing all worlds in \( e \) and retaining only those that agree sensing-wise with the real world \( w \).

**Definition 3.43 (Progression of an Epistemic State).** Let \( e \) be an epistemic state, \( w \) a world, and \( z \) a sequence of action standard names. The **progression of \( e \) through \( z \) wrt \( w \)**, denoted as \( e^z_w \), is given by

\[
e^z_w = \{ w'_z \mid w' \in e \text{ and } w' \simeq_z w \}
\] (3.130)

The progression of the single worlds \( w'_z \) above is according to Definition 3.35.
We are now ready to define the semantics of formulas:

**Definition 3.44.** Given an epistemic state $e \subseteq W$, a world $w \in W$, and a sentence $\alpha$, we define $e, w \models \alpha$ (read: “$e, w$ satisfies $\alpha$” or “$\alpha$ is true in $e, w$”) as $e, w, \langle \rangle \models \alpha$, where for any $z \in Z$:

1. $e, w, z \models F(t_1, \ldots, t_k)$ iff $w[F(n_1, \ldots, n_k), z] = 1$, where $n_i = |t_i|^z_w$;
2. $e, w, z \models (t_1 = t_2)$ iff $n_1$ and $n_2$ are identical, where $n_i = |t_i|^z_w$;
3. $e, w, z \models \alpha \land \beta$ iff $e, w, z \models \alpha$ and $e, w, z \models \beta$;
4. $e, w, z \models \neg \alpha$ iff $e, w, z \not\models \alpha$;
5. $e, w, z \models \forall x. \alpha$ iff $e, w, z \models \alpha(x)^n$ for all $n \in \mathcal{N}_x$;
6. $e, w, z \models [t] \alpha$ iff $e, w, z \cdot p \models \alpha$, where $p = |t|^z_w$;
7. $e, w, z \models \Box \alpha$ iff $e, w, z \cdot z' \models \alpha$ for all $z' \in Z$;
8. $e, w, z \models \forall P. \alpha$ iff $e, w', z \models \alpha$ for all $w' \sim_P w$;
9. $e, w, z \models \forall f. \alpha$ iff $e, w', z \models \alpha$ for all $w' \sim_f w$;
10. $e, w, z \models \text{Know}(\alpha)$ iff for all $w' \in e^w_z$, $e^w_z, w', \langle \rangle \models \alpha$;
11. $e, w, z \models \text{OKnow}(\alpha)$ iff for all $w'$, $w' \in e^w_z$ iff $e^w_z, w', \langle \rangle \models \alpha$.

A sentence $\alpha$ is called **satisfiable** if some $e$ and $w$ exist such that $e, w \models \alpha$. When $\Sigma$ is a set of sentences and $\alpha$ is a sentence, we write $\Sigma \models \alpha$ (read: “$\Sigma$ logically entails $\alpha$”) to mean that for every pair $e, w$, if $e, w \models \alpha'$ for every $\alpha' \in \Sigma$, then also $e, w \models \alpha$. Finally, we write $\models \alpha$ (read: “$\alpha$ is valid”) to mean $\{\} \models \alpha$.

Rules 1 to 7 are as in Definition 3.8. Rules 8 and 9 are furthermore exactly as in Definition 3.22.\(^5\)

The last two rules finally are the one that define the meaning of $\text{Know}$ and $\text{OKnow}$ according to possible world semantics, and including sensing compatibility with the actual world.

Just like the original $\mathcal{ES}$ introduced by Lakemeyer and Levesque [LL04], the logic that we defined here is a special case of possible-world semantics where knowledge is simply modelled as a **set** of worlds. Unsurprisingly, we thus obtain the properties of weak S5 [FHMV95], and

\(^5\)It was pointed to the author by Hector Levesque that this definition does not work in case we quantify a predicate from the outside into an epistemic operator, such as in the formula $\forall P \text{Know}(\forall x P(x))$. Since this case is of no relevance for this thesis, we simply stick to this definition. Otherwise, one possible solution is to resort to second-order variable maps that take a world as one of their arguments as is done in Lakemeyer and Levesque’s original formalization [LL09a].
since we use a fixed universe of discourse, the Barcan formula for knowledge and its existential version hold as well:

Theorem 3.45.

1. \(\models \Box \text{Know}(\alpha) \land \text{Know}(\alpha \supset \beta) \supset \text{Know}(\beta)\);
2. \(\models \Box \text{Know}(\alpha) \supset \text{Know}(\text{Know}(\alpha))\);
3. \(\models \Box \neg \text{Know}(\alpha) \supset \text{Know}(\neg \text{Know}(\alpha))\);
4. \(\models \Box \forall x \text{Know}(\alpha) \supset \text{Know}(\exists x \alpha)\);
5. \(\models \Box \exists x \text{Know}(\alpha) \supset \text{Know}(\exists x \alpha)\).

3.5.2 Regression

To extend regression to cope with subjective subformulas, we use the theorem below. It can be viewed as a successor state axiom for the Know operator, however it is a theorem of the logic rather than an axiom that we have to provide explicitly. Since this thesis uses a variant where actions sense standard names instead of truth values, a corresponding adapted version of Lakemeyer and Levesque’s original proof is provided as well.

Theorem 3.46. Let \(x\) be a variable that does not appear freely in \(\alpha\). Then

\[ (3.131) \quad \models \Box[a] \text{Know}(\alpha) \equiv \exists x. \text{sf}(a) = x \land \text{Know}(\text{sf}(a) = x \supset [a] \alpha) \]

Proof. \(\Rightarrow\): Let \(e, w, z \models [p] \text{Know}(\alpha_p)\) for any action name \(p\) and any \(z \in Z\). Let further \(w[\text{sf}(p), z] = n\). We have to show that \(e, w, z \models \text{Know}(\text{sf}(p) = n \supset [p] \alpha_p)\). Suppose \(w'_z \in e^w_z\), i.e. \(w' \in e\) and \(w' \equiv_z w\), and that \(w'_z \models \text{sf}(p) = n\). By Definition 3.35, \(w'[\text{sf}(p), z] = n\). But then \(w' \equiv_{z-p} w\) by Definition 3.42, hence \(e^w_{z-p}, w'_z \models \alpha_p\) by assumption, and therefore \(e^w_z, w'_z \models [p] \alpha_p\).

\(\Leftarrow\): Conversely, let \(e, w, z \models \exists x. \text{sf}(p) = x \land \text{Know}(\text{sf}(p) = x \supset [p] \alpha)\) for any action name \(p\) and any \(z \in Z\). Therefore there is some name \(n\) such that \(w[\text{sf}(p), z] = n\) and \(e, w, z \models \text{Know}(\text{sf}(p) = n \supset [p] \alpha)\). We need to show that \(e, w, z \models [p] \text{Know}(\alpha)\), that is \(e, w, z \cdot p \models \text{Know}(\alpha)\). Let \(w'_z \in e^w_{z-p}\), then \(w' \in e\) and \(w' \equiv_{z-p} w\). By Definition 3.42, this means that \(w' \equiv_z w\) and \(w'[\text{sf}(p), z] = w[\text{sf}(p), z] = n\). Hence \(w'_z \models (\text{sf}(p) = n)\), therefore by assumption \(e^w_z, w'_z, () \models [p] \alpha\), from which \(e^w_{z-p}, w'_z, () \models \alpha\) follows. \(\Box\)

We regress non-objective formulas with respect to a pair of basic action theories \(\Sigma'\) and \(\Sigma\). Intuitively, \(\Sigma'\) represents the beliefs of the agent, whereas \(\Sigma\) contains the laws by which the real world abides.
**Definition 3.47.** We define \( R[\alpha] \) as \( R[\Sigma', \Sigma, \langle \rangle, \alpha] \), where for any sequence \( \sigma \) of terms of sort action, \( R[\Sigma', \Sigma, \sigma, \alpha] \) is given as follows. Rules 1 – 10 are exactly as in Definition 3.30, but with the extra two arguments \( \Sigma' \) and \( \Sigma \), and where in the cases for the different types of fluent atoms, the axioms of \( \Sigma \) are used. We further have the following additional rule for \( \text{Know} \), where \( \kappa \) denotes the right-hand side of the equivalence in Theorem 3.46 (with a fresh variable \( x \)):

11. \( R[\Sigma', \Sigma, \sigma, \text{Know}(\alpha)] \) is defined inductively by:

\[
\begin{align*}
(a) \quad & R[\Sigma', \Sigma, \langle \rangle, \text{Know}(\alpha)] = \text{Know}(R[\Sigma', \Sigma', \langle \rangle, \alpha]); \\
(b) \quad & R[\Sigma', \Sigma, \sigma \cdot t, \text{Know}(\alpha)] = R[\Sigma', \Sigma, \sigma, \kappa]\cdot t.
\end{align*}
\]

The soundness of the new regression operator is given by:

**Theorem 3.48** (Lakemeyer and Levesque [LL04]). Let \( \Sigma \) and \( \Sigma' \) be basic action theories and \( \alpha \) be a bounded, basic sentence. Then \( R[\alpha] \) is static and satisfies

\[
\Sigma \land \text{OKnow}(\Sigma') \models \alpha \iff \Sigma_0 \land \text{OKnow}(\Sigma'_0) = R[\alpha].
\]

If we are only concerned about the agent’s internal view, we only need to regress wrt the BAT believed by the agent:

**Corollary 3.49.** Let \( \Sigma \) be a basic action theory over \( \langle D, F \rangle \) and \( \alpha \) a basic, bounded sentence that mentions only the fluents in \( D \cup F \). Then

\[
\models \text{OKnow}(\Sigma) \supset \text{Know}(\alpha) \iff \models \text{OKnow}(\Sigma_0) \supset \text{Know}(R[\Sigma, \Sigma, \langle \rangle, \alpha]).
\]

**Example 3.50.** Suppose we have a robot that has an action \( \text{lookFor}(x) \) at its disposal. The action is at the same time a sensing action, where its sensing result will be the location of object \( x \), as well as a physical action with the effect that in the process of searching object \( x \), the robot ends up at the location of \( x \). Let \( \Sigma_{\text{act}} \cup \Sigma_{\text{post}} \) hence consist of the following two axioms:

\[
(3.132) \quad \Box(sf(a) = y) \equiv \\
\quad \exists x. \ a = \text{lookFor}(x) \land y = \text{loc}(x) \lor \neg \exists x (a = \text{lookFor}(x) \land y = \text{ok})
\]

\[
(3.133) \quad \Box[a](\text{loc}(x) = y) \equiv \\
\quad x = \text{robot} \land \exists x' (a = \text{lookFor}(x') \land y = \text{loc}(x')) \lor \text{loc}(x) = y \land (x \neq \text{robot} \lor \neg \exists x' a = \text{lookFor}(x'))
\]

For simplicity, we do not include any precondition axiom. Now let \( \Sigma \) be the BAT consisting of

\[
(3.134) \quad \Sigma_0 = \{ \text{loc}(\text{book}) = \text{room6213}, \text{loc}(\text{cup}) = \text{kitchen} \}\]
3.5 Knowledge and Sensing

Together with (3.132) and (3.133), whereas \( \Sigma' \) is the union of
\[
\Sigma'_0 = \{ \text{loc(cup) = kitchen} \}
\]
and (3.132) and (3.133). If \( \Sigma' \) is the BAT believed by the robot, and \( \Sigma \) the one the actual world satisfies, this means that the robot has accurate knowledge about sensing and action effects as well as the location of the cup, but it is initially not aware of the location of the book.

Let now \( \alpha \) be the formula that expresses that the location of the book will be known after looking for it, that is
\[
\alpha = [\text{lookFor(book)}] \exists x \text{Know}(x = \text{loc(book)}).
\]

Let \( l \) be shorthand for \( \text{lookFor(book)} \). Then
\[
\mathcal{R}[\alpha] = \mathcal{R}[\Sigma, \Sigma', \langle \rangle, \alpha] = \mathcal{R}[\Sigma, \Sigma', \langle l \rangle, \exists x \text{Know}(x = \text{loc(book)})].
\]

Applying Theorem 3.46 yields
\[
\exists x. \mathcal{R}[\Sigma, \Sigma', \langle \rangle, \exists x'. x'(l) = x' \land \text{Know}(x'(l) = x' \supset [l](x = \text{loc(book)}))].
\]
Furthermore, \( sf(l) = x' \) regresses to \( x' = \text{loc(book)} \) both inside and outside of \( \text{Know} \), as \( \Sigma \) and \( \Sigma' \) agree on \( sf \). \( [l](x = \text{loc(book)}) \) regresses to \( x = \text{loc(book)} \) since looking for the book does not change its location. We end up with the (simplified) regression result of
\[
\exists x \exists x'. x' = \text{loc(book)} \land \text{Know}(x' = \text{loc(book)} \supset x = \text{loc(book)}).
\]
The above is obviously entailed by \( \Sigma_0 \land \text{OKnow}(\Sigma'_0) \) if we let \( x' = \text{room6213} = x \). Therefore, \( \Sigma \land \text{OKnow}(\Sigma') \models \alpha \).

As opposed to regression, we do not need a special form of progression in the epistemic case. The reason is that while regression transforms the query properties, progression operates on the agent’s knowledge base. Whereas queries now may contain knowledge operators that need to be handled by the regression operator, the agent’s initial knowledge usually is still a collection of objective sentences, and therefore can be progressed by the already available methods.

3.5.3 The Representation Theorem

While regression solves the projection problem for non-objective formulas by eliminating modal action operators, its end result may still contain modal epistemic operators. The idea behind Levesque and Lakemeyer’s Representation Theorem for OL [Lev84, LL01] is to recursively replace subjective subformulas of the form \( \text{Know}(\alpha) \) either by their truth value (wrt the agent’s
knowledge base), in case $\alpha$ is closed, or by an objective formula that represents the known instances of $\alpha$, in case it has free variables. For the latter, we can resort to the same “trick” that we already used for Universal Generalization in Section 3.2.3: When a formula contains a free variable, it suffices to check the entailment for a finite number of standard names, namely all those that are contained in the query and the knowledge base, plus one that does not appear and serves as representative for all non-mentioned names. Formally:

**Definition 3.51.** Let $\phi$ be an objective formula and $\Sigma_0$ be a finite set of objective sentences. Suppose that $n_1, \ldots, n_k$ are all the standard names in $\phi$ or in $\Sigma_0$, and that $n'$ is some name that does not appear in $\phi$ or in $\Sigma_0$. Then $\text{RES}[\phi, \Sigma_0]$ is defined by:

1. If $\phi$ has no free variables, then $\text{RES}[\phi, \Sigma_0]$ is $\top$, if $\Sigma_0 \models \phi$, and $\bot$, otherwise.

2. If $x$ is a free variable in $\phi$, then $\text{RES}[\phi, \Sigma_0]$ is

$$\left( (x = n_1) \land \text{RES}[\phi_{n_1}, \Sigma_0] \right) \lor \cdots$$
$$\left( (x = n_k) \land \text{RES}[\phi_{n_k}, \Sigma_0] \right) \lor$$
$$\left( (x \neq n_1) \land \cdots \land (x \neq n_k) \land \text{RES}[\phi_{n'}, \Sigma_0] \right).$$

Note that in the last case, we first substitute the new $n'$ for $x$, then recursively evaluate $\text{RES}$, after which we replace $n'$ again by $x$. Thus, the $n'$ is only a temporary placeholder for all unmentioned individuals and will not appear in the overall result. As we have discussed before, the trick does not work in the case of number variables. Therefore, throughout this section we assume that formulas do not mention any number terms.

Using the above, it is possible to define a recursive evaluation operator for arbitrary basic and static formulas as follows:

**Definition 3.52.** Given a finite set of static, objective sentences $\Sigma_0$ and a static, basic formula $\alpha$, $\| \alpha \|_{\Sigma_0}$ is defined by

1. $\| \alpha \|_{\Sigma_0} = \alpha$, when $\alpha$ is a static objective formula;

2. $\| \neg \alpha \|_{\Sigma_0} = \neg \| \alpha \|_{\Sigma_0}$;

3. $\| \alpha \land \beta \|_{\Sigma_0} = \| \alpha \|_{\Sigma_0} \land \| \beta \|_{\Sigma_0}$;

4. $\| \exists x \alpha \|_{\Sigma_0} = \exists x \| \alpha \|_{\Sigma_0}$;

5. $\| \text{Know}(\alpha) \|_{\Sigma_0} = \text{RES}[\| \alpha \|_{\Sigma_0}, \Sigma_0]$.

Then we have:
Theorem 3.53 (Lakemeyer and Levesque [LL04]). Let $\Sigma_0$ be a set of static, objective sentences and $\alpha$ a static, basic sentence. Then

$$|= \text{OKnow}(\Sigma_0) \supset \text{Know}(\alpha) \iff |= \Sigma_0 \supset |\alpha|_{\Sigma_0}.$$  

Example 3.54. Suppose that the robot believes the initial theory

$$\Sigma_0 = \{\text{loc}(\text{cup}) = \text{kitchen}, \exists x \ (\text{loc}(x) = \text{kitchen})\}$$

and we want to check whether only-knowing $\Sigma_0$ entails knowing some object in the kitchen:

$$\alpha = \exists x \text{Know}(\text{loc}(x) = \text{kitchen})$$

We have that

$$|\alpha|_{\Sigma_0} = \exists x \text{RES}[|\text{loc}(x) = \text{kitchen}|_{\Sigma_0}, \Sigma_0] = \exists x \text{RES}[\text{loc}(x) = \text{kitchen}, \Sigma_0].$$

The standard names in $\Sigma_0$ and $\alpha$ are cup and kitchen. Let $n'$ be a new standard name. RES[$\text{loc}(x) = \text{kitchen}, \Sigma_0$] therefore is the formula

$$(x = \text{cup}) \land \text{RES}[\text{loc}(\text{cup}) = \text{kitchen}, \Sigma_0] \lor$$

$$(x = \text{kitchen}) \land \text{RES}[\text{loc}(\text{kitchen}) = \text{kitchen}, \Sigma_0] \lor$$

$$(x \neq \text{cup}) \land (x \neq \text{kitchen}) \land \text{RES}[\text{loc}(n') = \text{kitchen}, \Sigma_0]_{n'}.$$  

Because neither $\text{loc}(\text{kitchen}) = \text{kitchen}$ nor $\text{loc}(n') = \text{kitchen}$ is entailed by $\Sigma_0$, the corresponding RES subformulas evaluate to $\bot$. $\text{loc}(\text{cup}) = \text{kitchen}$ however holds according to $\Sigma_0$, so we get

$$\text{RES}[\text{loc}(x) = \text{kitchen}, \Sigma_0] = (x = \text{cup}).$$

Intuitively, this accurately describes the known instances of $\text{loc}(x) = \text{kitchen}$. As the formula $\exists x (\text{loc}(x) = \text{kitchen})$ is trivially satisfied, we obtain $|= \text{OKnow}(\Sigma_0) \supset \text{Know}(\alpha)$.

As a final remark, note that if we drop $\text{loc}(\text{cup}) = \text{kitchen}$ from $\Sigma_0$, the entailment no longer holds. Even though we would still have $\exists x (\text{loc}(x) = \text{kitchen})$, the agent would only know that there is some object in the kitchen (de dicto), but it would not know its identity (de re).

It is moreover possible to combine the Representation Theorem with regression. For that matter, we first regress the query sentence, and afterwards apply $| \cdot |$. Thus, reasoning about the knowledge and action of an agent can be reduced to a finite number of instances of classical first-order theorem proving.

Theorem 3.55 (Lakemeyer and Levesque [LL04]). Given a pair of basic action theories $\Sigma$ and $\Sigma'$, and a bounded, basic sentence $\alpha$,

$$\Sigma \land \text{OKnow}(\Sigma') \models \alpha \iff \Sigma_0 \models |R[\alpha]|_{\Sigma_0}.$$
3.6 Golog

The theory we have developed so far allows us to represent and reason about an agent’s actions and knowledge. What we have ignored until now is the question of how the agent should choose its course of action in order to achieve a certain goal, based on what it knows about the state and the dynamics of its environment. A simple approach is to use planning: Assuming that the agent’s goal is given in form of a formula $\alpha$, we search for a ground action sequence leading to a situation where $\alpha$ is known to hold. This could be done, for instance, using iterative deepening search, where candidate sequences $\langle t_1, \ldots, t_k \rangle$ of increasing length $k$ are tested by means of projection:

$$\Sigma \models [t_1] \cdots [t_k]\alpha$$

For solving such projection tasks, we can then use one of the methods presented in Section 3.4. This approach however soon reaches its limits. On the one hand, we ignored the fact that the agent typically also has to employ its sensors in order to gather necessary information at runtime (i.e. while executing a previously planned course of action), which would require that instead of sequential plans we have to search for conditional plans [Lak99] that have the form of trees, which is even more costly. On the other hand, in our envisioned application scenario, the robot will typically perform a multitude of different tasks over an indefinite time period, and hence the search space, both in terms of branching factor and search depth, is usually way too large.

On the other hand, the human domain designer often has already some understanding of how a certain task is to be solved. The orthogonal approach to planning is therefore programming, which means in our case that we equip the agent not only with a basic action theory, but also some program that controls its behaviour:

```plaintext
while $\neg \forall x. \text{Room}(x) \supset \text{Clean}(x)$ do
  if $\neg \text{Clean}(\text{livingroom})$
    goto(\text{livingroom}); \text{clean(\text{livingroom})}
  endIf;
  if $\neg \text{Clean}(\text{kitchen})$
    goto(\text{kitchen}); \text{clean(\text{kitchen})}
  endIf
  ...
endWhile
```

(3.137)

On the one hand, such an agent program resembles the ones that we can write with common programming languages such as C++ or Java in the sense that it contains control structures like
conditionals (if) and loops (while). The difference on the other hand is its high-level nature: A Java program’s atomic instructions typically are variable assignments of the form \( x := y \times 2 \), the execution of which amounts to the CPU doing some calculation and then writing the updated value back into a register or memory cell. Our robot program’s basic building blocks though are the actions as defined in the basic action theory. In the highly abstracted perspective that we assume in this context, an action such as \( \text{goto(livingroom)} \) is considered atomic. The physical execution of it however is in itself a complex task that requires to solve a multitude of lower-level problems such as path planning, real-time collision avoidance, and sensor fusion.

Predefining the agent’s behaviour by means of programming has the drawback that the programmer must be able to foresee every eventuality that the agent may encounter in its lifetime. In particular in the case that the robot performs an open-ended, indefinite task this is of course unrealistic. The agent programming language Golog (for alGOL in LOGic) was introduced [LRL+97] to allow for a middle ground between the two extremes of pure planning and pure programming: In Golog, the programmer can freely combine deterministic, imperative constructs such as while and if with nondeterministic aspects, such as nondeterministic choice of branches, of arguments, or of number of iterations. Thus, it is possible to manually adjust the mixture between planning and programming, and concentrate the search effort to where it is actually needed. For example, instead of solving the problem of cleaning all rooms by a blind search over all action sequences, we may define the following program:

\[
(\pi x. (\text{Room}(x) \land \neg \text{Clean}(x))?; \text{goto}(x); \text{clean}(x))^*; (\forall y. \text{Room}(y) \supset \text{Clean}(y))?
\]

Intuitively, it means to nondeterministically pick (\( \pi \)) some instantiation for the variable \( x \) such that \( \text{Room}(x) \land \neg \text{Clean}(x) \) holds, then perform the action \( \text{goto}(x) \), followed by \( \text{clean}(x) \). This is to be iterated a nondeterministically chosen number of times greater than or equal to zero (\( ^* \)) such that afterwards \( \forall y. \text{Room}(y) \supset \text{Clean}(y) \) comes to hold. Such a program can be viewed as a plan sketch whose gaps then have to be filled by the system through search. In the above example, the search space is constrained by forcing a successful execution to be a sequence of the form

\[
\text{goto}(n_1); \text{clean}(n_1); \text{goto}(n_2); \text{clean}(n_2); \ldots
\]

In particular, this disallows any execution where two clean actions are performed consecutively, or two goto actions etc., leaving only the order of the rooms in the solution to be determined by the agent.
3.6.1 Simple Golog

The original Golog language, as introduced by Levesque et al. [LRL+97] defines the semantics of programs meta-theoretically through macro expansion. Lakemeyer and Levesque later showed that a similar definition can be given within ES, and that the two are (in a certain formal sense) equivalent [LL05a]. They define a macro \( \text{Do}(\delta, \alpha) \) that takes a program \( \delta \) and a formula \( \alpha \) as arguments, and expands to a formula that expresses under which circumstances there is a possible successful execution of \( \delta \) after which \( \alpha \) comes to hold:

**Definition 3.56** (The \( \text{Do} \) macro). \( \text{Do} \) is defined inductively as follows:

- **Atomic action:**
  \[
  \text{Do}(t, \alpha) \overset{\text{def}}{=} \text{Poss}(t) \land [t]\alpha \tag{3.139}
  \]

- **Test:**
  \[
  \text{Do}(\phi? , \alpha) \overset{\text{def}}{=} \phi \land \alpha \tag{3.140}
  \]

- **Sequence:**
  \[
  \text{Do}(\delta; \delta', \alpha) \overset{\text{def}}{=} \text{Do}(\delta, \text{Do}(\delta', \alpha)) \tag{3.141}
  \]

- **Nondeterministic choice:**
  \[
  \text{Do}(\delta|\delta', \alpha) \overset{\text{def}}{=} \text{Do}(\delta, \alpha) \lor \text{Do}(\delta', \alpha) \tag{3.142}
  \]

- **Nondeterministic choice of argument:**
  \[
  \text{Do}(\pi x . \delta, \alpha) \overset{\text{def}}{=} \exists x . \text{Do}(\delta, \alpha) \tag{3.143}
  \]

- **Iteration:**
  \[
  \text{Do}(\delta^\ast, \alpha) \overset{\text{def}}{=} \forall P . \{ \Box (\alpha \supset P) \land \Box (\text{Do}(\delta, P) \supset P) \} \supset P \tag{3.144}
  \]

where \( P \) is some predicate neither appearing in \( \alpha \) nor in \( \delta \).

For example, if \( \delta \) is \( \pi x . (\text{Room}(x) \land \neg \text{Clean}(x))?; \text{goto}(x); \text{clean}(x) \), then \( \text{Do}(\delta, \alpha) \) expands to

\[
\exists x . \text{Room}(x) \land \neg \text{Clean}(x) \land \text{Poss}(\text{goto}(x)) \land [\text{goto}(x)](\text{Poss}(\text{clean}(x)) \land [\text{clean}(x)]\alpha)
\]
Furthermore, conditionals and loops can be defined in terms of the above constructs, and therefore are considered “syntactic sugar”:

\[
\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endIf} \overset{\text{def}}{=} [\phi?; \delta_1][\neg\phi?; \delta_2]
\]

\[
\text{while } \phi \text{ do } \delta \text{ endwhile} \overset{\text{def}}{=} [\phi?; \delta]^*; \neg\phi?
\]

In the following, we will refer to the set of programs that can be constructed from the above mentioned constructs as simple Golog.

### 3.6.2 ConGolog

The original Golog was later generalized to ConGolog by De Giacomo et al. [GLL00]. It includes the following extensions:

- **Concurrency:** If \( \delta_1 \) and \( \delta_2 \) are programs, then \( \delta_1 |\delta_2 \) denotes their concurrent execution. Here and in the following, concurrency is always to be understood as interleaving the primitive actions performed by the two subprograms. That is to say that ultimately the execution of concurrent processes boils down to a (not necessarily strictly alternating) sequence of actions drawn from the involved subprograms.

- **Prioritized Concurrency:** The construct \( \delta_1 \rangle\rangle \delta_2 \) is similar to \( \delta_1 |\delta_2 \) with the additional requirement that \( \delta_1 \) has higher priority, i.e. \( \delta_2 \) can only proceed its execution when \( \delta_1 \) is currently blocked or finished.

- **Concurrent Iteration:** Similar as to how \( \delta^* \) may be interpreted as “perform \( \delta \) zero or more times in sequence”, \( \delta^1 \) stands for “perform \( \delta \) zero or more times concurrently”.

- **Interrupts:** The intuition behind an interrupt \( < \phi \rightarrow \delta > \) is that \( \delta \) is triggered once condition \( \phi \) becomes true. Interrupts can be defined in terms of other constructs:

\[
< \phi \rightarrow \delta > \overset{\text{def}}{=} \text{while } \text{InterruptsRunning} \text{ do if } \phi \text{ then } \delta \text{ else } \perp? \text{ endwhile}
\]

Here, \( \text{InterruptsRunning} \) is a special fluent that is set to true by the special action \( \text{startInterrupts} \), and set to false by \( \text{stopInterrupts} \). To execute a program \( \delta' \) containing interrupts, we then actually execute

\[
\text{startInterrupts} ; (\delta' \rangle\rangle \text{stopInterrupts} ).
\]

- **Exogenous Actions:** Finally, there may happen changes in the agent’s environment that are not due to the agent itself. To model such dynamic environments, it is possible to
define a set of actions that may not be used within a user-defined program, but that are viewed to be under the control of “nature”. Suppose that this is done through a special fluent \( \text{Exo}(a) \) for which we have a corresponding axiom in the \( \Sigma_{\text{def}} \) of the basic action theory. We then have a special program:

\[
\delta_{\text{exo}} \overset{\text{def}}{=} ((\pi a)(\text{Exo}(a)?; a))^*
\]

To analyze how executing the agent’s program \( \delta \) interacts with the dynamic environment, we then simply have to study the possible executions of \( \delta || \delta_{\text{exo}} \).

De Giacomo et al. furthermore present a new semantics that is based on single-step transitions. For that matter, they use a predicate \( \text{Trans}(\delta, s, \delta', s') \) whose intuitive meaning is that when \( \delta \) is the current program and \( s \) the current situation, then there is a transition such that \( \delta' \) becomes the remaining program and \( s' \) is the resulting situation. A transition may either be a primitive action \( t \), in which case \( s' = \text{do}(t, s) \), or a test \( \phi? \), in which case \( s = s' \). Another predicate \( \text{Final}(\delta, s) \) then expresses that \( \delta \) may legally terminate in \( s \). They provide an axiomatization of these predicates that ensures that the program constructs have the desired meaning, and that is moreover compatible with the macro semantics of the original GOLOG. Below, we will provide a corresponding semantics for programs in \( \mathcal{ES} \).

### 3.6.3 IndiGolog

While ConGOLOG already provides a rich amount of expressiveness to define agent behaviour, there are two issues that make it problematic for real-world applications. On the one hand, programs are executed offline only: This means that the program interpreter first analyzes the entire input program in order to find a conforming execution sequence before the first action is actually done. This soon becomes a problem, in particular for larger programs. Moreover, especially scenarios where the agent has incomplete knowledge about its environment make it necessary to gather information at runtime through sensing. IndiGOLOG [GL99, GLLS04] is yet another extension to GOLOG that tackles both these issues.

IndiGOLOG executes programs online, which means that there is no general lookahead; the interpreter simply executes the next possible action in each step, treating nondeterminism like random choices. A new operator \( \Sigma(\delta) \) is introduced which has to be used to explicitly mark subprograms which have to be solved by means of search. This does not represent a loss of generality since one might still encapsulate the entire program within \( \Sigma(\cdot) \), but it gives the programmer much more control over where the system spends its computational effort. In addition, programs may contain sensing actions for acquiring needed information at runtime.
When such an action is executed, a sensing result is obtained (which normally is the current value of some fluent) and used to update the agent’s knowledge base. Thus, a subsequent choice in the program that depends on this sensed value can be made online.

The semantics of IndiGolog is defined through appropriately extending the transition semantics of ConGolog, which lends itself particularly well for the online execution of programs.

3.6.4 Golog in ES

The variant of Golog to be employed in this thesis draws its inspiration from all of the above. Its syntax and semantics is defined as follows:

**Definition 3.57 (Golog).** A *program* \( \delta \) is any expression that can be formed from the constructs provided in Figure 3.2. Additionally, we include the abbreviations \( nil \overset{\text{def}}{=} \top \) for the empty program that successfully terminates without doing any action, and \( fail \overset{\text{def}}{=} \bot \) for the program that always fails. A *configuration* \( (z, \delta) \) then consists of an action sequence \( z \in Z \) and a program \( \delta \). Intuitively, \( z \) is the history of actions that have already been performed, while \( \delta \)
1. \((z, \alpha?) \in \mathcal{F}_{e,w}\) if \(e, w, z \models \alpha\);

2. \((z, \delta_1; \delta_2) \in \mathcal{F}_{e,w}\) if \((z, \delta_1) \in \mathcal{F}_{e,w}\) and \((z, \delta_2) \in \mathcal{F}_{e,w}\);

3. \((z, \delta_1|\delta_2) \in \mathcal{F}_{e,w}\) if \((z, \delta_1) \in \mathcal{F}_{e,w}\) or \((z, \delta_2) \in \mathcal{F}_{e,w}\);

4. \((z, \pi x.\delta) \in \mathcal{F}_{e,w}\) if \((z, \delta^x_n) \in \mathcal{F}_{e,w}\) for some \(n \in \mathbb{N}\);

5. \((z, \delta^\ast) \in \mathcal{F}_{e,w}\);

6. \((z, \delta_1||\delta_2) \in \mathcal{F}_{e,w}\) if \((z, \delta_1) \in \mathcal{F}_{e,w}\) and \((z, \delta_2) \in \mathcal{F}_{e,w}\);

7. \((z, \rho\delta_1|\delta_2) \in \mathcal{F}_{e,w}\) if \((z, \rho\delta_1) \in \mathcal{F}_{e,w}\) and \((z, \delta_2) \in \mathcal{F}_{e,w}\);

8. \((z, \delta^l) \in \mathcal{F}_{e,w}\);

9. \((z, \Sigma(\delta)) \in \mathcal{F}_{e,w}\) if \((z, \delta) \in \mathcal{F}_{e,w}\);

10. \((z, \Gamma(\alpha, \delta)) \in \mathcal{F}_{e,w}\) if \(e, w, z \not\models \alpha\) or \((z, \delta) \in \mathcal{F}_{e,w}\).

Figure 3.3: Program Finality

is the program that remains to be executed.

Given an epistemic state \(e\) and a world \(w\), the \textit{finality predicate} \(\mathcal{F}_{e,w}\) over configurations is the least set that satisfies the rules presented in Figure 3.3. Furthermore, the \textit{transition relation} \((z, \delta) \xrightarrow{e,w} (z', \delta')\) among configurations is the least set satisfying the rules depicted in Figure 3.4, where \(\xrightarrow{e,w}^\ast\) refers to the reflexive transitive closure of \(\xrightarrow{e,w}\).

There are a few noteworthy differences to existing variants of GOLOG.

1. First, the above semantics is meta-theoretic rather than being defined through an axiomatization within the logic. This not only helps to keep the definition simple\(^6\), but will also come in handy in Chapter 5 when we embed it into the semantics of an extension of \(\mathcal{ES}\) that allows to express properties about programs.

2. Next, in our definition, there are no test transitions. Instead, transitions are always physical actions, and tests are merely viewed as \textit{conditions} for doing actions or terminating

\(^6\)Among other things, De Giacomo et al. have to reify programs as terms in their semantics for \textsc{ConGolog}, which is particularly troublesome in light of the fact that programs may contain formulas. Hence, all predicates, functions and logical connectives have to be reified as well.
1. $\langle z, t \rangle \xrightarrow{e,w} \langle z \cdot p, nil \rangle$, if $p = |t|_n$;

2. $\langle z, \delta_1; \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \gamma; \delta_2 \rangle$, if $\langle z, \delta_1 \rangle \xrightarrow{e,w} \langle z \cdot p, \gamma \rangle$;

3. $\langle z, \delta_1; \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$, if $\langle z, \delta_1 \rangle \in \mathcal{F}^{e,w}$ and $\langle z, \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

4. $\langle z, \delta_1; \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$, if $\langle z, \delta_1 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$ or $\langle z, \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

5. $\langle z, \pi x. \delta \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$, if $\langle z, \delta^x \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$ for some $n \in N_x$;

6. $\langle z, \delta^n \rangle \xrightarrow{e,w} \langle z \cdot p, \gamma; \delta^n \rangle$, if $\langle z, \delta \rangle \xrightarrow{e,w} \langle z \cdot p, \gamma \rangle$;

7. $\langle z, \{Env ; \delta \} \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$, if $\langle z, \delta^P_{\{Env; \delta \}} \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

8. $\langle z, \{Env : P_i(\delta) \} \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$, if $\langle z, \{Env ; \delta^i \} \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

9. $\langle z, \delta_1 | \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, (\delta' | \delta_2) \rangle$, if $\langle z, \delta_1 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

10. $\langle z, \delta_1 | \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, (\delta' | \delta_2) \rangle$, if $\langle z, \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

11. $\langle z, \delta_1 | \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, (\delta' | \delta_2) \rangle$, if $\langle z, \delta_1 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$;

12. $\langle z, \delta_1 | \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, (\delta_1 | \delta') \rangle$, if $\langle z, \delta_2 \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$ and there is no $\gamma, p'$ such that $\langle z, \delta_1 \rangle \xrightarrow{e,w} \langle z \cdot p', \gamma \rangle$;

13. $\langle z, \delta' \rangle \xrightarrow{e,w} \langle z \cdot p, (\gamma | \delta') \rangle$, if $\langle z, \delta \rangle \xrightarrow{e,w} \langle z \cdot p, \gamma \rangle$;

14. $\langle z, \Sigma(\delta) \rangle \xrightarrow{e,w} \langle z \cdot p, \Sigma(\delta') \rangle$, if $\langle z, \delta \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$ and $\langle z \cdot p, \delta' \rangle \xrightarrow{e,w} \langle z \cdot p \cdot z', \delta'' \rangle$ and $\langle z \cdot p \cdot z', \delta'' \rangle \in F^{e,w}$;

15. $\langle z, \Gamma(\alpha, \delta) \rangle \xrightarrow{e,w} \langle z \cdot p, \Gamma(\alpha, \delta') \rangle$, if $e, w, z \models \alpha$ and $\langle z, \delta \rangle \xrightarrow{e,w} \langle z \cdot p, \delta' \rangle$.

Figure 3.4: Program Transitions
execution. This has certain advantages, but also possibly drawbacks. We will discuss this issue more thoroughly in Section 5.1.3.

3. Our definition works both in a purely objective setting as well as for knowledge-based programs [Rei01b, CL06a]. The latter refers to the case when we want to encode an agent that possesses incomplete knowledge about its environment and has to employ sensing actions to gather necessary information at runtime. Test conditions then are subjective formulas such as \( \neg \text{Know(In(bob, kitchen))} \), which are evaluated with respect to an epistemic state.

4. Since it is completely sufficient for the purposes of this thesis, we use a simple variant of the search operator \( \Sigma \) that only returns sequential plans. Note that in general, this may be insufficient, in particular in the case of an agent with incomplete knowledge that uses sensing. We may rather have to also worry about the epistemic feasibility [Lev96, SGLL04] of plans and programs: Consider a conditional statement \( \text{if } \phi \text{ then } a \text{ else } b \text{ endIf} \). If \( a \) and \( b \) are both executable, then no matter whether \( \phi \) holds or not, there is some successful execution of the program. Nonetheless, it may not be epistemically feasible, namely when the agent does not know if \( \phi \) holds, and it hence does not know which of the two branches to take. To ensure that the program can indeed be executed physically, it has to be ensured that the agent in due time comes to know whether \( \phi \) holds, for instance by sensing \( \phi \) prior to testing it.

5. We additionally defined the operator \( \Gamma(\alpha, \delta) \) for guarded execution. Intuitively, it executes \( \delta \) only as long as the condition \( \alpha \) holds. This for instance allows to have programs react to dynamic changes in the environment happening during the execution of a program, e.g. due to exogenous actions.

### 3.7 A Knowledge-Based Agent

We now have all the ingredients for building a knowledge-based agent. Our formalization provides the following meta-level operations:

1. **Initialization**: Before the agent starts operating, we provide it with some initial world knowledge, in particular regarding the preconditions and effects of its actions as well as the meaning of sensing results.

2. **Update**: When executing an action physically, incorporate its sensing result. We presume that the agent chooses its actions according to some online executed \textsc{Gолог} program.
3. **Query:** Check whether a given formula is known to be true. The query can either be a user request or a test condition to be evaluated within the agent’s control program.

4. **Input:** Provide the agent with additional, new information.

We presume that our agent operates in a classical sense-reason-act cycle: First, the program interpreter evaluates the currently remaining Golog program to determine the next action, which may involve solving the projection task for test conditions in the program (“reason”). Physically executing the chosen action (“act”) yields some sensing result to be incorporated in the knowledge base (“sense”), and a new cycle begins. At any time during its operation, human users may furthermore interact with the robot by providing it with new knowledge or requesting information from it.

The interaction with the agent is via an expressive query language that includes non-objective formulas that can encode facts such as “there is an unknown object in the room.” As we will see in this section, it suffices, using the techniques introduced in this chapter, to represent the agent’s beliefs through a finite, objective knowledge base. Semantically, the state of the system is given by a tuple of the form \( \langle e, w, \sigma \rangle \), where \( e \) represents the agent’s knowledge, \( w \) the actual world, and \( \sigma \) the history of action performed so far. The above interaction operations are then formally defined as follows:

**Definition 3.58** (Interaction Operations). Let \( \Sigma \) be a set of objective sentences, \( e \subseteq \mathcal{W} \), \( w \in \mathcal{W} \), \( t \) a primitive term or standard name of sort \texttt{action}, \( \sigma \) a sequence of such terms, and \( \alpha \) a sentence. Then

1. \( \text{INIT}(\Sigma, w) = \langle e, w, \langle \rangle \rangle \),
   where \( e = \{ w' \mid w' \simeq w, w' \models \Sigma \} \);

2. \( \text{EXE}(\langle e, w, \sigma \rangle, t) = \langle e', w, \sigma \cdot t \rangle \),
   iff \( e' = \{ w' \in e \mid w' \models [\sigma](\text{sf}(t) = n) \} \) and \( w \models [\sigma](\text{sf}(t) = n) \);

3. \( \text{ASK}(\langle e, w, \sigma \rangle, \alpha) = \text{“yes”} \)
   iff \( e, w' \models [\sigma]\alpha \) for all \( w' \in e \) (and “no” otherwise);

4. \( \text{TELL}(\langle e, w, \sigma \rangle, \alpha) = \langle e', w, \sigma \rangle \),
   iff \( e' = \{ w' \in e \mid e, w' \models [\sigma]\alpha \} \).

The above can be viewed as a generalization of the ASK and TELL operations Levesque and Lakemeyer suggest for interacting with a static knowledge base [Lev84, LL01], inspired by similar operations defining abstract data types like stacks and queues. The theorems we discuss below similarly generalize their corresponding theorems to the above set of operations.
Note the requirement that the action to be executed is a primitive term or standard name. We thus forbid that the agent attempts to perform an insufficiently specified action such as \( \text{goto}(\text{loc}(\text{bestfriend}(\text{ann}))) \). It is rather forced to first determine the identity of \( \text{bestfriend}(\text{ann}) \), given in the form of some standard name \( n \). It then moreover needs to identify \( \text{loc}(n) \) by some standard name \( n' \), and only then is able to send the command \( \text{goto}(n') \) to the execution module.

### 3.7.1 Regression

First, consider an agent that uses purely regression-based reasoning. Our objective is to represent the system’s state by a collection of purely standard first-order formulas and reduce the execution of our interaction operations to standard first-order theorem proving. As an intermediate step, the following theorem will show us how our meta-theoretic definition of the operations translates to a characterization by valid ES sentences.

If \( \Sigma \) is a set of objective sentences and \( w \in W \), let \( \mathcal{R}[\Sigma]_w = \{ w' \mid w' \simeq w, w' \models \Sigma \} \).

Intuitively, \( e = \mathcal{R}[\Sigma]_w \) is the epistemic state represented by the formulas \( \Sigma \) if primitive action terms are interpreted according to \( w \). Observe that according to the semantics, \( \mathcal{R}[\Sigma]_w \) is the unique epistemic state such that \( e, w \models \text{OKnow}(\Sigma) \).

**Theorem 3.59.** Let \( \Sigma \) be a basic action theory, \( \alpha \) a sentence, \( \phi \) a fluent sentence, \( t \) a primitive term or standard name of sort action, \( \sigma \) a sequence of such terms, and \( w \in W \). Then

1. \( \text{INIT}(\Sigma, w) = (\mathcal{R}[\Sigma]_w, w, \langle \rangle) \);
2. \( \text{EXE}(\mathcal{R}[\Sigma]_w, w, \sigma, t) = (\mathcal{R}[\Sigma \land \phi]_w, w, \sigma \cdot t) \) iff \( \models \text{OKnow}(\Sigma) \supset \text{Know}([\sigma](sf(t) = n) \equiv \phi) \) and \( w \models [\sigma](sf(t) = n) \);
3. \( \text{ASK}(\mathcal{R}[\Sigma]_w, w, \sigma, \alpha) = \text{"yes"} \) iff \( \models \text{OKnow}(\Sigma) \supset \text{Know}([\sigma]\alpha) \);
4. \( \text{TELL}(\mathcal{R}[\Sigma]_w, w, \sigma, \alpha) = (\mathcal{R}[\Sigma \land \phi]_w, w, \sigma) \) iff \( \models \text{OKnow}(\Sigma) \supset \text{Know}([\sigma] \alpha \equiv \phi) \).

**Proof.**

1. Immediate from Definition 3.58.
2. \( \text{EXE}(\mathcal{R}[\Sigma]_w, w, \sigma) = (\mathcal{R}[\Sigma \land \phi]_w, w, \sigma \cdot t) \) iff \( \{ w' \in \mathcal{R}[\Sigma]_w \mid w' \models [\sigma](sf(t) = n) \} = \{ w' \mid w' \simeq w, w' \models \Sigma \land \phi \} \), where \( w \models [\sigma](sf(t) = n) \) (by Definition 3.58).
3.7 A Knowledge-Based Agent

Theorem 3.58

iff \( \{ w' \in R[\Sigma]_w \mid w' = [\sigma](sf(t) = n) \} = \{ w' \in R[\Sigma]_w \mid w' = \phi \} \),

where \( w = [\sigma](sf(t) = n) \) (by definition of \( R \))

iff for all \( w' \in R[\Sigma]_w \), \( w' = [\sigma](sf(t) = n) \) iff \( w' = \phi \),

where \( w = [\sigma](sf(t) = n) \) (equivalent rewriting)

iff \( OKnow(\Sigma) \supset Know([\sigma](sf(t) = n) \equiv \phi) \),

where \( w = [\sigma](sf(t) = n) \) (by the semantics)

3. ASK((\( R[\Sigma]_w, w, \sigma \)), \( \alpha \)) = “yes”

iff \( R[\Sigma]_w, w' = [\sigma]\alpha \) for all \( w' \in R[\Sigma]_w \) (by Definition 3.58)

iff \( OKnow(\Sigma) \supset Know([\sigma]\alpha) \) (by the semantics)

4. TELL((\( R[\Sigma]_w, w, \sigma \)), \( \alpha \)) = (\( R[\Sigma \land \phi]_w, w, \sigma \))

iff \( R[\Sigma \land \phi]_w = \{ w' \in R[\Sigma]_w \mid R[\Sigma]_w, w' = [\sigma]\alpha \} \) (by Definition 3.58)

iff \( \{ w' \mid w' \preceq_{(\_)} w, w' = \Sigma \land \phi \} = \{ w' \in R[\Sigma]_w \mid R[\Sigma]_w, w' = [\sigma]\alpha \} \) (by definition of \( R \))

iff \( \{ w' \in R[\Sigma]_w \mid w' = \phi \} = \{ w' \in R[\Sigma]_w \mid R[\Sigma]_w, w' = [\sigma]\alpha \} \) (by definition of \( R \))

iff for all \( w' \in R[\Sigma]_w \), \( w' = \phi \) iff \( R[\Sigma]_w, w' = [\sigma]\alpha \) (equivalent rewriting)

iff \( OKnow(\Sigma) \supset Know(\phi \equiv [\sigma]\alpha) \) (by the semantics) \( \Box \)

The theorem tells us that the initial state of the system can simply be represented by the initial action theory itself together with an empty action history. After an action \( t \) was executed and a sensing result provided, it suffices to augment the current representation \( \Sigma \) by a fluent sentence that expresses the necessary and sufficient conditions under which that sensing result is to be expected, and add \( t \) to the action history. A basic, bounded query \( \alpha \) can be answered by testing whether only-knowing the action theory entails currently knowing \( \alpha \). If the system is told some sentence \( \alpha \) about the current situation \( \sigma \), the knowledge base is to be augmented by a fluent sentence that is known to be equivalent to \([\sigma]\alpha\). Notice that for a fluent sentence \( \phi \) and a basic action theory \( \Sigma \), the theory \( \Sigma \land \phi \) is also a basic action theory when we view \( \phi \) as part of the new \( \Sigma_\phi \). Therefore, if we start in a state given by \( INIT(\Sigma, w) \), successively applying TELL and EXE always leads to a state that is itself representable by some action theory \( \Sigma' \), and we may pose queries using ASK. Achieving the final reduction of states and operations to first-order reasoning now is straightforward considering the techniques presented in Section 3.5: We may eliminate actions using regression, and can treat knowledge by means of \( \| \cdot \|_{\Sigma_\alpha} \), thus acquiring
the $\phi$ in question in the above theorem. Using this idea, we obtain the main result of this section in form of representation theorems for the semantic definition of our interaction operations:

**Theorem 3.60.** Let $\Sigma$ be a basic action theory over $\langle D, F \rangle$, $\alpha$ a basic, bounded sentence mentioning only fluents from $D \cup F$, $t$ a primitive term or standard name of sort action, $\sigma$ a sequence of such terms, and $w \in W$. Let $R[\sigma, \alpha]$ denote $R[\Sigma, \Sigma, \sigma, \alpha]$. Then

1. $\text{INIT}(\Sigma, w) = \langle R[\Sigma]_w, w, \langle \rangle \rangle$;
2. $\text{EXE}(\langle R[\Sigma]_w, w, \sigma \rangle, t) = \langle R[\Sigma \land R[\sigma, sf(t) = n]]_w, w, \sigma \cdot t \rangle$ iff $w \models [\sigma](sf(t) = n)$;
3. $\text{ASK}(\langle R[\Sigma]_w, w, \sigma \rangle, \alpha) = \text{"yes"}$ iff $w \models [\Sigma_0 \supset R[\sigma, \alpha]]_{\Sigma_0}$;
4. $\text{TELL}(\langle R[\Sigma]_w, w, \sigma \rangle, \alpha) = \langle R[\Sigma \land R[\sigma, \alpha]]_{\Sigma_0} \rangle_w, w, \sigma \rangle$.

**Proof.**

1. Immediate from Definition 3.58.

2. $\text{EXE}(\langle R[\Sigma]_w, w, \sigma \rangle, t) = \langle R[\Sigma \land R[\sigma, sf(t) = n]]_w, w, \sigma \cdot t \rangle$
   
   iff $w \models [\sigma](sf(t) = n)$
   
   and $w \models [\sigma](sf(t) = n)$ (by Theorem 3.59)

   iff $w \models [\sigma](sf(t) = n)$
   
   and $w \models [\sigma](sf(t) = n)$ (by Corollary 3.49)

   iff $\Sigma_0 \supset [R[\sigma, (sf(t) = n)] \equiv R[\sigma, sf(t) = n]]_{\Sigma_0}$
   
   and $w \models [\sigma](sf(t) = n)$ (by Theorem 3.53)

   iff $\Sigma_0 \supset (R[\sigma, sf(t) = n] \equiv R[\sigma, sf(t) = n])$
   
   and $w \models [\sigma](sf(t) = n)$ (by definition of $R$)

   iff $\Sigma_0 \supset \top$ and $w \models [\sigma](sf(t) = n)$ (by the semantics)

   iff $w \models [\sigma](sf(t) = n)$ (by the semantics)

3. $\text{ASK}(\langle R[\Sigma]_w, w, \sigma \rangle, \alpha) = \text{"yes"}$

   iff $\models OKnow(\Sigma) \supset Know([\sigma]_{\alpha})$ (by Theorem 3.59)

   iff $\models OKnow(\Sigma_0) \supset Know(R[\sigma, \alpha])$ (by Corollary 3.49)

   iff $\Sigma_0 \supset [R[\sigma, \alpha]]_{\Sigma_0}$ (by Theorem 3.53)
4. \( TELL(⟨ℜ[Σ]_w, w, σ⟩, α) = ⟨ℜ[Σ ∧ R[σ, α]|Σ_0]_w, w, σ⟩ \)
   \( \text{iff} \ | = OKnow(Σ) ⊃ Know([σ]|Σ_0) \) (by Theorem 3.59)
   \( \text{iff} \ | = OKnow(Σ_0) ⊃ Know(⟨⟩, [σ]|Σ_0) \) (by Corollary 3.49)
   \( \text{iff} \ Σ_0 ⊃ [R[⟨⟩]|σ, α]|Σ_0 \equiv [R[σ, α]|Σ_0] \) (by definition of \( R \) and \( ||·||_Σ_0 \))
   \( \text{iff} \ Σ_0 ⊃ ⊤ \) (which is a tautology, therefore the proposition holds) □

**Example 3.61.** As an example of how an agent according to the above formalization operates, consider the following program for the office robot:

\[
\textbf{while} \ \neg ∃ x \text{Know}(x = \text{loc(book)}) \ \textbf{do} \ \text{lookFor(book)} \ \textbf{endWhile}
\]

The above is what we call a knowledge-based program [Rei01b, CL06a], which refers to a GOLOG program executed by an agent with incomplete world knowledge that employs sensing to gather necessary information at runtime. Syntactically, this manifests itself of course in the facts that the program contains sensing actions and that all its test conditions are subjective formulas that only talk about the robot’s beliefs.

Suppose that the robot’s basic action theory consists of the sensing axiom and successor state axiom we used in Example 3.50:

(3.150) \[ \Box(sf(a) = y) \equiv \exists x. a = \text{lookFor}(x) ∧ y = \text{loc}(x) \lor \neg ∃ x(a = \text{lookFor}(x) ∧ y = \text{ok}) \]

(3.151) \[ \Box[a](\text{loc}(x) = y) \equiv x = \text{robot} \land ∃ x'(a = \text{lookFor}(x') ∧ y = \text{loc}(x')) \]
\[ \lor \text{loc}(x) = y ∧ (x ≠ \text{robot} \lor ∃ x'a = \text{lookFor}(x')) \]

We assume that the robot’s initial theory \( Σ_0 \) is empty, i.e. it does not possess any initial knowledge about the state of the world. Let \( w \) be the actual world.

1. If \( Σ \) denotes the above stated BAT, we start with initializing the agent, yielding
   \[ \text{INIT}(Σ, w) = ⟨ℜ[Σ]_w, w, ⟨⟩⟩. \]

2. In an online execution of the above program, the first thing to do is to evaluate whether the loop condition holds. We therefore call
   \[ \text{ASK}(⟨ℜ[Σ]_w, w⟩, \neg ∃ x \text{Know}(x = \text{loc(book)})). \]
According to Theorem 3.60, this reduces to checking whether

$$\models \Sigma_0 \supset [R(\langle \rangle, \neg \exists x \text{Know}(x = \text{loc}(\text{book})))]|\Sigma_0.$$ 

In this case, regression through $\langle \rangle$ leaves the formula unchanged. Moreover,

$$\models \neg \exists x \text{Know}(x = \text{loc}(\text{book})))|\Sigma_0 = \neg \exists x \text{RES}[x = \text{loc}(\text{book}), \Sigma_0],$$

which reduces to $\neg \exists x \bot$ since there are no known locations of $\text{book}$ according to the empty $\Sigma_0$. The answer is hence “yes” and the program subsequently enters the body of the while loop.

3. Let $l$ be shorthand for $\text{lookFor}(\text{book})$. The system now calls

$$\text{EXE}(\langle R[\Sigma], w, \langle \rangle, l \rangle).$$

Assume $w$ returns $\text{room6213}$ as sensing result, i.e. $w \models sf(l) = \text{room6213}$. To apply Theorem 3.60, we need to determine $R[\langle \rangle, sf(l) = \text{room6213}]$, which by (3.150) and using simplification is $\text{room6213} = \text{loc}(\text{book})$. If $\Sigma'$ stands for $\Sigma \land \text{room6213} = \text{loc}(\text{book})$, we thus end up in the state

$$\langle R[\Sigma'], w, \langle l \rangle \rangle.$$ 

4. Next, the loop condition needs to be evaluated once again, which is done by the call

$$\text{ASK}(\langle R[\Sigma'], w, \langle l \rangle \rangle, \neg \exists x \text{Know}(x = \text{loc}(\text{book}))).$$

Theorem 3.60 again states that this is the same as checking

$$\models \Sigma_0' \supset [R[\langle l \rangle, \neg \exists x \text{Know}(x = \text{loc}(\text{book})))]|\Sigma_0',$$

where $\Sigma_0'$ is $\text{room6213} = \text{loc}(\text{book})$. Similar as in Example 3.50, applying regression leaves us with determining whether

$$\models \Sigma_0 \supset [\neg \exists x \exists x'. x' = \text{loc}(\text{book}) \land \text{Know}(x' = \text{loc}(\text{book}) \supset x = \text{loc}(\text{book}))]|\Sigma_0',$$

which amounts to checking if

$$\models \Sigma_0 \supset \neg \exists x \exists x'. x' = \text{loc}(\text{book}) \land \text{RES}[x' = \text{loc}(\text{book}) \supset x = \text{loc}(\text{book}), \Sigma_0].$$

Since the location of the $\text{book}$ is known to be $\text{room6213}$, the argument of RES is entailed by $\Sigma_0'$ just in case $x'$ is not $\text{room6213}$ or $x$ is $\text{room6213}$. Therefore, the resulting first-order theorem proving task is deciding whether

$$\models \Sigma_0 \supset \neg \exists x \exists x'. x' = \text{loc}(\text{book}) \land (x' \neq \text{room6213} \lor x = \text{room6213}),$$
3.7 A Knowledge-Based Agent

which is equivalent to
\[ \models \neg \exists x \ x = room6213. \]

The overall answer to the query is hence “no” and the program interpreter quits the loop, which finishes the program.

5. Finally suppose that the human user tells the robot that the cup is in the kitchen. This translates to a call of the form
\[ \text{TELL}(⟨ℜ[Σ′]w, w, ⟨l⟩⟩, \text{loc}(cup) = \text{kitchen}). \]

By Theorem 3.60, we have to add
\[ [R[⟨l⟩], \text{loc}(cup) = \text{kitchen}]_{Σ′0} \]

5. Finally suppose that the human user tells the robot that the cup is in the kitchen. This translates to a call of the form
\[ \text{TELL}(⟨ℜ[Σ′]w, w, ⟨l⟩⟩, \text{loc}(cup) = \text{kitchen}). \]

By Theorem 3.60, we have to add
\[ [R[⟨l⟩], \text{loc}(cup) = \text{kitchen}]_{Σ′0} \]

to the agent’s knowledge base. Because \text{lookFor}(\text{book}) has no effect on the location of the cup, regression leaves the formula unchanged. Furthermore, \[ [\cdot]_{Σ′0} \] has no effect on the formula either since it is objective. The new knowledge base of the agent thus becomes
\[ Σ′′_0 = Σ_0 \wedge \text{loc}(cup) = \text{kitchen}, \]
and the resulting system state is
\[ ⟨ℜ[Σ′′_0]w, w, ⟨l⟩⟩. \]

3.7.2 Progression

In Section 3.4.2 we already argued that in real-world applications it is not practical to entirely rely on regression as the means for projection. Since the agent’s history of executed actions will keep growing during its lifetime and soon becomes unmanageable for regression, we would rather have to regularly update the agent’s initial knowledge base such that it reflects the then current state of the world. That is, we take a somewhat mixed approach where updates after the physical execution of actions are done using progression, whereas lookahead and reasoning about future situations still rely on regression.

As opposed to the regression case, system states are now not represented by a basic action theory whose initial knowledge base refers to the initial situation, but by a theory of the form \[ [σ]Σ, \] where \Σ is a basic action theory and \σ the history of actions performed so far. First, we need this lemma:

**Lemma 3.62.** Let \Σ be a set of objective sentences, \α a basic sentence and \σ a sequence of action standard names and/or primitive terms. Then
\[ \models O\text{Know}(σ|Σ) ⊃ K\text{now}(σ|α) \iff \models O\text{Know}(Σ) ⊃ K\text{now}(α). \]
Theorem 3.63. Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ and $\Sigma' = \Sigma'_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be (first-order) basic action theories with the same definitional and successor state axioms, $\alpha$ a sentence, $\phi$ a fluent sentence, $t$ a primitive term or standard name of sort action, $\sigma$ a sequence of such terms, and $w \in W$. Further suppose that $\Sigma'_0$ is a (first-order) progression of $\Sigma_0 \land \phi$ through $t$ wrt $\Sigma_{\text{def}} \cup \Sigma_{\text{post}}$. Then

1. $\text{INIT}(\Sigma, w) = \langle \Re[\Sigma]_w, w, \langle \rangle \rangle$;

2. $\text{EXE}(\langle \Re[[\sigma]_\Sigma]_w, w, \sigma, t \rangle) = \langle \Re[[\sigma \cdot t]\Sigma']_w, w, \sigma \cdot t \rangle$
   iff $\models \text{OKnow}(\Sigma) \cup \text{Know}(\sigma sf(t) = n \equiv \phi)$ and $w \models [\sigma sf(t) = n]$ (by Definition 3.58)

3. $\text{ASK}(\langle \Re[[\sigma]_\Sigma]_w, w, \sigma, \alpha \rangle) = \langle \Re[[\sigma \land \phi]]_w, w, \sigma \rangle$
   iff $\models \text{OKnow}(\Sigma) \cup \text{Know}(\alpha \equiv \phi)$.

Proof.

1. Immediate from Definition 3.58.

2. $\text{EXE}(\langle \Re[[\sigma]_\Sigma]_w, w, \sigma, t \rangle) = \langle \Re[[\sigma \cdot t]\Sigma']_w, w, \sigma \cdot t \rangle$
   iff $\{w' \models [\sigma sf(t) = n] \mid w' \models [\sigma sf(t) = n]\} = \{w' \models [\sigma sf(t) = n] \mid w' \models [\sigma sf(t) = n]\}$
   where $w \models [\sigma sf(t) = n]$ (by Definition 3.58)

   iff $\{w' \models [\sigma sf(t) = n] \mid w' \models [\sigma sf(t) = n]\} = \{w' \models [\sigma sf(t) = n] \mid w' \models [\sigma sf(t) = n]\}$
   where $w \models [\sigma sf(t) = n]$ (by definition of $\Re$)

   iff for all $w' \models [\sigma sf(t) = n]$, $w' \models [\sigma sf(t) = n]$ iff $w' \models [\sigma sf(t) = n]$
   where $w \models [\sigma sf(t) = n]$ (equivalent rewriting)

   iff for all $w' \models [\sigma sf(t) = n]$, $w' \models [\sigma sf(t) = n]$, $w' \models [\sigma sf(t) = n]$
   where $w \models [\sigma sf(t) = n]$ (by Definition 3.35)

   iff for all $w' \models [\sigma sf(t) = n]$, $w' \models [\sigma sf(t) = n]$
   where $w \models [\sigma sf(t) = n]$ (by Definition 3.37)
Above we slightly abused the notation of progressed worlds introduced in Definition 3.35: When writing $w_z$, we actually mean $w_z$ such that $z$ is the sequence of coreferring standard names for the terms in $\sigma$. Note that since $\sigma$ contains only standard names or at most primitive terms of sort action, and due to our assumption that actions are rigid, the denotation of terms in $\sigma$ is the same in any situation $z'$.

Similar to the regression case, we can implement the interaction operations with the help of regression and the Representation Theorem. Note that regression is still needed: In the EXE case, we need it to replace the sensing result formula $(sf(t) = n)$ by the right-hand side of the

$\iff$ for all $w' \simeq_0 \sigma$, $w' \models [\sigma](\Sigma \land sf(t) = n)$ iff $w' \models [\sigma](\Sigma \land \phi)$,
where $w \models [\sigma](sf(t) = n)$ (by Definition 3.35)

$\iff$ for all $w' \simeq_0 w$ with $w' \models [\sigma]$, $w' \models [\sigma](sf(t) = n)$ iff $w' \models [\sigma] \phi$,
where $w \models [\sigma](sf(t) = n)$ (by Definition 3.35)

$\iff \models OKnow([\sigma] \supset \text{Know}([\sigma](sf(t) = n \equiv \phi)))$,
where $w \models [\sigma](sf(t) = n)$ (by the semantics)

$\iff \models OKNow(\Sigma) \supset \text{Know}(sf(t) = n \equiv \phi)$,
where $w \models [\sigma](sf(t) = n)$ (by Lemma 3.62)

3. ASK$(\langle \mathcal{R}[\sigma] \Sigma w, w, \sigma \rangle, \alpha) = \text{"yes"}$

$\iff \mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \alpha$ for all $w' \in \mathcal{R}[\sigma] \Sigma w$ (by Definition 3.58)

$\iff \models OKnow([\sigma] \supset \text{Know}([\sigma] \alpha))$ (by the semantics)

$\iff \models OKNow(\Sigma) \supset \text{Know}(\alpha)$ (by Lemma 3.62)

4. TELL$(\langle \mathcal{R}[\sigma] \Sigma w, w, \sigma \rangle, \alpha) = \langle \mathcal{R}[\sigma](\Sigma \land \phi) w, w, \sigma \rangle$

$\iff \mathcal{R}[\sigma](\Sigma \land \phi) w = \{w' \in \mathcal{R}[\sigma] \Sigma w \mid \mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \alpha\}$ (by Definition 3.58)

$\iff \{w' \mid w' \simeq_0 \sigma, w' \models [\sigma] \Sigma \land \phi\}$

$= \{w' \in \mathcal{R}[\sigma] \Sigma w \mid \mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \alpha\}$ (by definition of $\mathcal{R}$)

$\iff \{w' \in \mathcal{R}[\sigma] \Sigma w \mid w' \models [\sigma] \phi\}$

$= \{w' \in \mathcal{R}[\sigma] \Sigma w \mid \mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \alpha\}$ (by definition of $\mathcal{R}$)

$\iff$ for all $w' \in \mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \phi$ iff $\mathcal{R}[\sigma] \Sigma w, w' \models [\sigma] \alpha$ (equivalent rewriting)

$\iff \models OKnow([\sigma] \supset \text{Know}([\sigma](\phi \equiv \alpha)))$ (by the semantics)

$\iff \models OKNow(\Sigma) \supset \text{Know}(\phi \equiv \alpha)$ (by Lemma 3.62)
sensing axiom. Moreover, the argument $\alpha$ for TELL and ASK may still contain $[\cdot]$ operators. It is however not necessary to regress through the entire history of already executed actions, but only through those that are part of the query.

**Theorem 3.64.** Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ and $\Sigma' = \Sigma_0' \cup \Sigma_{\text{def}}' \cup \Sigma_{\text{post}}$ be (first-order) basic action theories with the same definitional and successor state axioms over $\langle D, F \rangle$, $\alpha$ a basic, bounded sentence mentioning only fluents from $D \cup F$, $t$ a primitive term or standard name of sort action, $\sigma$ a sequence of such terms, and $w \in W$. Let $R[\alpha]$ denote $R[\Sigma, \Sigma, \langle \cdot \rangle, \alpha]$. Then

1. $\text{INIT}(\Sigma, w) = \langle R[\Sigma]w, w, \langle \cdot \rangle \rangle$;
2. $\text{EXE}((R[\sigma]w, w, \sigma), t) = \langle R[\sigma[t]w, w, \sigma \cdot t) \iff w \models \sigma(sf(t) = n)
   \quad \text{and } \Sigma_0' \text{ is a (first-order) progression of } \Sigma_0 \land R[sf(t) = n] \text{ through } t \text{ wrt } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$.
3. $\text{ASK}(R[\sigma]w, w, \sigma), \alpha) = \text{“yes” } \iff \Sigma_0 \supset \|R[\alpha]\|_{\Sigma_0};$
4. $\text{TELL}(R[\sigma]w, w, \sigma), \alpha) = \langle R[\sigma]w, w, \sigma \rangle$.

**Proof.**

1. Immediate from Definition 3.58.

2. $\text{EXE}((R[\sigma]w, w, \sigma), t) = \langle R[\sigma[t]\Sigma']w, w, \sigma \cdot t)
   \quad \iff \models \text{OKnow}(\Sigma) \supset \text{Know}(sf(t) = n \equiv R[sf(t) = n])
   \quad \text{where } \Sigma_0' \text{ is a progression of } \Sigma_0 \land R[sf(t) = n] \text{ through } t \text{ wrt } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$
   \quad \text{and } $w \models \sigma(sf(t) = n)$
   \quad \text{(by Theorem 3.63)}$
   \quad \iff \models \text{OKnow}(\Sigma_0) \supset \text{Know}(R[sf(t) = n \equiv R[sf(t) = n]])
   \quad \text{where } \Sigma_0' \text{ is a progression of } \Sigma_0 \land R[sf(t) = n] \text{ through } t \text{ wrt } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$
   \quad \text{and } $w \models \sigma(sf(t) = n)$
   \quad \text{(by Corollary 3.49)}$
   \quad \iff \models \Sigma_0 \supset \|R[sf(t) = n \equiv R[sf(t) = n]|_{\Sigma_0}
   \quad \text{where } \Sigma_0' \text{ is a progression of } \Sigma_0 \land R[sf(t) = n] \text{ through } t \text{ wrt } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$
   \quad \text{and } $w \models \sigma(sf(t) = n)$
   \quad \text{(by Theorem 3.63)}$
   \quad \iff \models \Sigma_0 \supset (R[sf(t) = n] \equiv R[sf(t) = n])$
   \quad \text{where } \Sigma_0' \text{ is a progression of } \Sigma_0 \land R[sf(t) = n] \text{ through } t \text{ wrt } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$
   \quad \text{and } $w \models \sigma(sf(t) = n)$
   \quad \text{(by definition of } R)$
3.7 A Knowledge-Based Agent

iff \( \models \Sigma_0 \supset \top \),
\( \Sigma'_0 \) is a progression of \( \Sigma_0 \land \mathcal{R}[sf(t) = n] \) through \( t \) wrt \( \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \)
and \( w \models [\sigma](sf(t) = n) \) (by the semantics)
iff \( \Sigma'_0 \) is a progression of \( \Sigma_0 \land \mathcal{R}[sf(t) = n] \) through \( t \) wrt \( \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \)
and \( w \models [\sigma](sf(t) = n) \) (by the semantics)

3. \( \text{ASK}(\langle\mathcal{R}[\sigma]\Sigma\rangle_w, w, \sigma), \alpha) = \text{"yes"} \)
iff \( \models O\text{Know}(\Sigma) \supset \text{Know}(\alpha) \) (by Theorem 3.63)
iff \( \models O\text{Know}(\Sigma_0) \supset \text{Know}(\mathcal{R}[\alpha]) \) (by Corollary 3.49)
iff \( \models \Sigma_0 \supset \|\mathcal{R}[\alpha]\|_{\Sigma_0} \) (by Theorem 3.53)

4. \( \text{TELL}(\langle\mathcal{R}[\sigma]\Sigma\rangle_w, w, \sigma), \alpha) = \langle\mathcal{R}[\sigma](\Sigma \land \|\mathcal{R}[\alpha]\|_{\Sigma_0})\rangle_w, w, \sigma\rangle \)
iff \( \models O\text{Know}(\Sigma) \supset \text{Know}(\alpha \equiv \|\mathcal{R}[\alpha]\|_{\Sigma_0}) \) (by Theorem 3.63)
iff \( \models O\text{Know}(\Sigma_0) \supset \text{Know}(\mathcal{R}[\alpha \equiv \|\mathcal{R}[\alpha]\|_{\Sigma_0}]) \) (by Corollary 3.49)
iff \( \Sigma_0 \supset \|\mathcal{R}[\alpha \equiv \|\mathcal{R}[\alpha]\|_{\Sigma_0}]\|_{\Sigma_0} \) (by Theorem 3.53)
iff \( \Sigma_0 \supset (\|\mathcal{R}[\alpha]\|_{\Sigma_0} \equiv \|\mathcal{R}[\alpha]\|_{\Sigma_0}) \) (by definition of \( \mathcal{R} \) and \( \| \cdot \|_{\Sigma_0} \))
iff \( \models \Sigma_0 \supset \top \) (which is a tautology, therefore the proposition holds) \( \square \)

Example 3.65. To illustrate the progression-based agent, consider the robot from Example 3.61 again, and suppose we want to execute the same program.

1. Initialization is exactly as before:
\[
\text{INIT}(\Sigma, w) = \langle\mathcal{R}\Sigma\rangle_w, w, \langle\rangle\rangle.
\]

2. The first evaluation of the loop condition is also exactly as in Example 3.61 due to the fact that Theorems 3.60 and 3.64 coincide when the history \( \sigma \) is empty. The answer to
\[
\text{ASK}(\langle\mathcal{R}\Sigma\rangle_w, w, \langle\rangle\rangle, \lnot\exists x \text{Know}(x = \text{loc}(\text{book})))
\]
is therefore “yes”.

3. To handle the call
\[
\text{EXE}(\langle\mathcal{R}\Sigma\rangle_w, w, \langle\rangle\rangle, l)
\]
where \( \text{room6213} \) is the sensing result, we need \( \mathcal{R}[sf(l) = \text{room6213}] \), which again simplifies to \( \text{room6213} = \text{loc}(\text{book}) \). Recall that we assumed an initially empty \( \Sigma_0 \), which is why
we only need to determine the progression of \{room6213 = loc(book)\} through \(l\) wrt \(\Sigma_{def} \cup \Sigma_{post}\). According to the SSA (3.151), \textit{lookFor(book)} does not change the value of \(loc(book)\). What however does change is the location of the robot, which becomes the same as the location of the object that is searched. Therefore the progression is

\[
\Sigma'_0 = \{\text{loc(robot)} = \text{loc(book)}, \text{loc(book)} = \text{room6213}\}.
\]

If \(\Sigma' = \Sigma'_0 \cup \Sigma_{def} \cup \Sigma_{post}\), the resulting system state hence is

\[
\langle \mathcal{R}[\Sigma']_w, w, (l)\rangle.
\]

4. Next, the loop condition needs to be evaluated once again, which is done by the call

\[
\text{ASK}(\langle \mathcal{R}[\Sigma']_w, w, (l)\rangle, \neg \exists x \text{Know}(x = \text{loc(book)})).
\]

Theorem 3.64 states that this is the same as checking

\[
\vdash \Sigma'_0 \supset [\mathcal{R}[\neg \exists x \text{Know}(x = \text{loc(book))}]]_{\Sigma'_0}\]

Observe that in contrast to the regression case, we here do not have to regress through \(l\), hence regression leaves the above formula unchanged. It remains to be checked whether

\[
\vdash \Sigma'_0 \supset \neg \exists x \text{RES}[x = \text{loc(book)}, \Sigma'_0].
\]

As the location of the book is known to be \textit{room6213}, this reduces to

\[
\vdash \Sigma'_0 \supset \neg \exists x. x = \text{room6213}.
\]

The consequence of the implication is obviously unsatisfiable, therefore we get “no” as the overall answer, forcing the program interpreter to quit the loop and finish the program, as before.

5. If the human user again tells the robot that the cup is in the kitchen, we are faced with a call of the form

\[
\text{TELL}(\langle \mathcal{R}[\Sigma']_w, w, (l)\rangle, \text{loc(cup)} = \text{kitchen}).
\]

By Theorem 3.64, we have to add

\[
[\mathcal{R}[\text{loc(cup)} = \text{kitchen}]]_{\Sigma'_0}
\]

to the agent’s knowledge base. As the formula is both static and objective, neither \(\mathcal{R}\) nor \(\cdot [\Sigma'_0]\) has any effect on it. We simply add it to the existing knowledge base, yielding \(\Sigma''_0 = \Sigma'_0 \land \text{loc(cup)} = \text{kitchen}\) and the new system state being

\[
\langle \mathcal{R}[\Sigma''_0]_w, w, (l)\rangle.
\]
3.8 Discussion

3.8.1 Summary

In this chapter, the logical foundations for building a knowledge-based agent were established. We argued that it is reasonable to adopt the idealized perspective that the agent is a logically omniscient first-order reasoner, and that the modal Situation Calculus variant $\mathcal{ES}$ is particularly suited as underlying logic. It was then shown how basic action theories can be used to encode dynamic domains, and that regression and progression are two possible means for solving the projection problem, which means reducing reasoning about actions and future situations to standard first-order theorem proving.

Furthermore, when we include epistemic operators in order to represent the sensing and knowledge of an agent, a similar reduction to first-order reasoning is possible via Lakemeyer and Levesque’s Representation Theorem. The GOLOG language moreover allows to define complex actions (i.e. programs) that include imperative and nondeterministic parts, thus enabling us to intermingle programming and planning. Finally, we discussed how all of the above can be put together to implement a knowledge-based agent.

There are many important issues involved with the design and implementation of (physical) agents that are beyond the scope of this thesis and were hence not addressed at all. First, real-world applications typically come with some kind of uncertainty: Information provided by sensors is usually noisy, actuators work in a certain margin of error, and intended actions may simply fail or not yield the desired outcome. Decision-theoretic variants of GOLOG such as DTGOLOG [BRST00] and READYLOG [FL08] amalgamate the language with stochastic actions and utilities; a corresponding formalization based on $\mathcal{ES}$ is presented by Ziegelmayer [Zie06]. A more fundamental approach is to integrate probabilities directly into the Situation Calculus [BHL95] or $\mathcal{ES}$ [GL07, BL11].

Another issue that we completely neglect in this thesis, but that is all the more important, is that an autonomous robot with an open-ended task should be capable of learning [Mit97]. Ideally, the learning process is integrated with the high-level control of the agent and accounts for uncertainty [BL12]. Furthermore, if an agent operates within a multi-agent scenario [HL01], we may need to explicitly represent other agents and their beliefs [BL10a, BL10b]. Moreover, in the context of an embodied agent such as an autonomous robot, there is the fundamental problem of symbol grounding [Har90], which, stated simply, corresponds to the question of how patterns of subsymbolic sensory data (say camera images of a cup of coffee) relate to the robot’s internal symbolic representation (e.g. the standard name cup). Somewhat related to this is the issue of how fuzzy, qualitative notions such as “close to” and “far from” are to be coped with at
the symbol level [SFL11, SFL12b]. Finally, for actually building an autonomous domestic robot one has to integrate low-level components such as localization, navigation, face recognition and natural language processing with the high-level control within a software framework [NFBL10] and a hardware platform [SFL12a].

3.8.2 Comparison to Other Formalizations

Since its original introduction by Lakemeyer and Levesque in 2004 [LL04], authors have used different variants of the logic $\mathcal{ES}$. We want to briefly discuss in what aspects the definition presented in this chapter, which is in fact yet another such variant, differs and where it coincides with the ones in existing literature.

Terms and Standard Names: Originally, Lakemeyer and Levesque [LL04] did not make a distinction between terms and standard names in $\mathcal{ES}$, and also not even between sorts. It was simply assumed that the (non-sorted) universe of discourse consists of the set of all ground terms, which still allows for a substitutional interpretation of quantification. The semantics definition thus was extremely simple and elegant and allowed to focus on the innovative aspects of the new logic.

However, apart from the fact that it is thus not possible to have the agent being ignorant about the identity of a term such as $\text{bestfriend}(\text{bob})$, the application of the Representation Theorem becomes problematic as we have to exclude function symbols in order to be still able to enumerate all individuals in the knowledge base and the query formula. Since actions typically have parameters, additional machinery is necessary to remove them from regression results [CL06a]. Here, we therefore resort to a definition similar to Lakemeyer and Levesque’s later formalization [LL05a] that includes different sorts and where worlds may disagree on the denotation of terms. One important consequence of this is the requirement mentioned in Section 3.7 that in order for an action to be epistemically feasible, it has to be given in the form of a primitive term or a standard name.

Progression: The semantical account of the progression of a theory used in an earlier publication [CELN07] was the $\mathcal{ES}$ counterpart of an updated definition of progression Reiter gave in his book [Rei01a]. It turned out that this new variant is actually weaker than the original one [LR97] in the sense that a BAT can have multiple progressions that are not logically equivalent, as pointed out by Vassos, Lakemeyer and Levesque [VLL08]. Section 3.4.2 therefore introduces a new definition that is based on Lakemeyer and Levesque’s notion of progressed worlds [LL09a], and Theorem 3.39 establishes the correspondence to Lin and Reiter’s original version.
Action Executability: The role of action preconditions has changed since Reiter presented his solution to the frame problem [Rei91]. Originally, Poss served as a guard for the SSA:

$$\text{Poss}(a, s) \supset [F(\vec{x}, \text{do}(a, s)) \equiv \gamma_F^+ \lor F(\vec{x}, s) \land \neg \gamma_F^-]$$

The value of fluents is thus only defined for situations that are the result of a possible action. Later formulations however typically drop the Poss atom in front of the SSA, as we did here. Often, this is simply a matter of keeping the formalism as simple as possible. As long as we only let the agent execute possible actions, we simply do not care about what values fluents have in situations that are unreachable anyway. Nevertheless, there is also a somewhat more philosophical justification for this.

We can distinguish two different types of possibility. On the one hand, there is physical possibility, such as in "it is not possible to walk through a wall". Lakemeyer and Levesque [LL10] take this view in their recent definition of $\mathcal{ES}$, where the sensing compatibility relation $w' \simeq_{z} w$ includes the requirement that the action $p$ is possible in $z$ according to $w'$. The agent thus comes to believe that Poss$(p)$ was indeed true after executing $p$, even if it was not in reality.

We take a different perspective in this thesis: We define the act of doing an action as the agent’s decision or commitment to it, which manifests itself once it sends the corresponding control signals to its actuators. In this sense, actions are always possible, only the effects that they have may differ, and it lies in the domain designer’s responsibility to appropriately reflect them in the successor state axioms. For example, if the robot believes it is standing one meter away from the wall, when in fact it is only 20 centimeters away, and it decides to do the action forward$\langle 100 \rangle$, then it tells its wheel actuators to roll forward by one meter. What will happen is that the robot indeed rolls the .2 meters towards the wall and, when it touches the wall, keeps spinning its wheels for a little while. The effect of that action in that case is that the robot ends up right in front of the wall. Depending on the circumstances, effects may differ, e.g. if the wall is thin and the robot is very strong, the wall may crush etc. In any case, we can represent these effects through appropriate SSAs.

In the latter scenario, the Poss predicate merely serves as another means of defining or restricting the agent’s behaviour, similar (or in addition) to what the GOLOG program does. This can be for reasons of safety (e.g. the robot should not attempt to move forward by one meter if its distance is less than that because that may cause damage), or Poss could also act as some sort of search control (e.g. it makes no sense to consider a putdown action if the robot is not holding anything). In both cases, it is conceivable to perform an action yet believe it was not “possible” in the sense that it was not safe or not targeted towards the agent’s goal.
Chapter 4

Planning in Golog

In the previous chapter we established ES and GOLOG as appropriate means for the representation and control of a logic-based agent. Furthermore it was shown that the central reasoning task of projection, be it by means of regression or by progression, can be reduced to the task of theorem proving within standard first-order logics. Thus, in principle, to implement such an agent, all that is needed is a GOLOG interpreter together with an FOL theorem prover.

However, evaluation shows [Gry10] that a naive implementation of this approach is only practicable in terms of expressiveness, but very poor in performance. The reason is that although GOLOG is a suitable means for the overall control of the agent, including sensing updates and plan monitoring, it nevertheless performs bad on certain tasks which involve a large amount of planning. Recall that in Section 3.6 we argued that GOLOG’s nondeterministic constructs allow to put as much or as little nondeterminism into a program as desired, and that thus the full range between deterministic programming and pure, uninformed planning can be exploited by the domain designer. For example, assume that our office robot has the task to serve coffee to all people in the office who want some. A (highly nondeterministic) control program in this case might look as follows:

\[
\text{while } \exists x (\text{WantsCoffee}(x) \land \neg \text{HasCoffee}(x)) \text{ do }
\]
\[
\pi a[A\text{ppropriate}(a)\text{?}; a]
\]
\[
\text{endWhile}
\]

The program says that as long as there exists a person \( x \) who wants coffee and has not yet received any, the robot should nondeterministically choose some appropriate action \( a \) and execute it. What constitutes an appropriate action in this context can be constrained by accordingly defining the \text{Appropriate}(a)\) predicate; in the simplest case, it simple admits every executable action, or all actions from a restricted subset. Any successful execution trace of such a program
gio
g


dailyRoutine()
makeCoffee();
achieve(∀x (WantsCoffee(x) ⊃ HasCoffee(x)));
getMail();
achieve(∀x (Letter(x) ⊃ On(x, desk(Addressee(x))));
...
endProc

Figure 4.1: Control procedure for the daily routine of an office robot

will then consist of a sequence of actions that, when executed, lead to a situation in which the goal formula ¬∃x (WantsCoffee(x) ∧ ¬HasCoffee(x)) holds. Solving this program through lookahead thus is nothing else than sequential planning in the classical sense. In fact, we can plan for any desired formula ϕ in a similar manner:

(4.2) while ¬ϕ do πa[Appropriate(a)?; a] endwhile

Let in the following achieve(ϕ) denote this program. We have argued that the GOLOG language allows the programmer to restrain the search space by including domain-dependent knowledge about what the solution to a certain task looks like, and how a problem divides into subproblems. While in many cases, specifying and implementing an agent in this manner is what makes the task feasible in the first place, for most physical agents, there are nevertheless certain subtasks which can hardly or not at all be constrained or subdivided any further. Typically, these are subproblems that are rather combinatorial in nature, which is to say that in these cases it is clear what needs to be done and which action operators are appropriate to fulfill the task, but it has to be figured out by the system what the right order of execution is. It is in these cases that we have to resort to something like our achieve(ϕ) subprogram above. Typical instances of this are scheduling currently pending requests to the system, finding a route through a certain topology, or combinations of these two. For example, our office robot may have a daily routine as sketched out in Figure 4.1. Although that procedure’s overall structure is quite deterministic, it nonetheless contains subproblems where planning has to be applied. After going to the coffee machine and making coffee, the robot may be required to bring coffee to people at different locations in the building (possibly at specific times), and therefore needs to plan for a route and schedule that ensures that all such requests get fulfilled in time, preferably using as few actions (or as little time and energy) as possible. Similarly, after going to the mail inbox and fetching all incoming mail, the letters need to be brought to their corresponding recipients’ desks, which
again involves planning for a corresponding route.

There is a huge variety of special purpose planners that outperform any existing GOLOG interpreter when it comes to such planning tasks. This is not at all surprising in light of the fact that GOLOG interpreters such as IndiGOLOG [GLLS04] or ReadyLog [FL08] usually do their lookahead by means of simple blind search, whereas state-of-the-art planners resort to a variety of sophisticated techniques and heuristics to enhance their performance.

In fact, despite their common origin, research on action logics on the one hand and automated planning on the other hand have developed rather independently in the past. This is mainly due to the fact that work on action languages focused on formalisms of high expressiveness, whereas planning researchers were concerned with their systems' computational efficiency. Of course, there is a trade-off between the two, which is why the usage of highly expressive languages such as GOLOG comes at the cost of limited efficiency, while efficient planners such as FF [HN01] require an input language with restricted expressiveness.

During the last one and a half decades however, one could observe that the two fields began to converge again. Exemplary for this trend is the development of the planning domain definition language PDDL [GHK+98], which has served as the input language for systems participating at the biennial international planning competition IPC [McD00], and which extends simple STRIPS-based planning by features such as conditional effects, time, concurrency, plan constraints and preferences. By now, PDDL has become a de-facto standard for the description of planning domains and problems, and it is understood by a multitude of state-of-the-art planners which have been developed since then.

It seems more than natural to try to exploit these achievements from the field of planning and apply them in the context of GOLOG, with the aim of thus acquiring a system which is both expressive and able to efficiently solve planning problems. The approach we will follow here is to embed existing PDDL planners into GOLOG, where the idea is that whenever the GOLOG interpreter encounters a planning subtask in the form of (4.2), it translates it into a PDDL problem, then calls the planner on this problem, and continues its execution on the returned solution plan.

In order for this approach to be sound, we first have to establish a common semantical basis to ensure that a plan returned by the PDDL planner is also a legal plan within ES for the situation from which the planner was called. We will do this by showing how to map a PDDL domain to a basic action theory that admits the same set of legal plans. Although it may seem that a translation in the opposite direction is what we actually need to embed a PDDL planner into GOLOG, there are at least two reasons in favor of this approach.

On the one hand, it is simpler. The reason is that PDDL is far more restricted in expres-
siveness than \( \mathcal{ES} \). While \( \mathcal{ES} \) basic action theories can use the full power of first-order logic, PDDL descriptions, among other things, contain the domain-closure and closed-world assumptions, i.e. it is assumed that the universe of discourse is finite, that all individuals are known, and that the complete current state is given through a set of positive literals. If we wanted to represent instances of the more expressive formalism by means of the less expressive one, we would have to find and apply restrictions that ensure that the input is still representable in the target language. On the other hand, the opposite is much more straightforward: Here, we will simply construct a basic action theory \( \Sigma = \text{Map}(P) \) for a given PDDL problem \( P \) that admits the same plans. We then use the inverse operation \( P = \text{Map}^{-1}(\Sigma) \) of this mapping for embedding the PDDL planner into GoLog. This means that we assume that the basic action theory used by the GoLog system is in a form that could result as the output of the \( \text{Map} \) operation, and that in calls to the planner, a corresponding PDDL problem is reconstructed. Technically, this does not guarantee that one covers the largest possible class of action theories that can be translated to PDDL. For this purpose, it is necessary to also analyze possible translations in the backward direction, which is done formally in [ENLC06], [RN07], and [RHN08] for the ADL subset of PDDL. Those results, which we will discuss in more detail at the end of Section 4.2 below, basically show that though \( \text{Map}^{-1} \) does not fully cover all translatable BATs, it is nonetheless a very close approximation.

On the other hand, the mapping can also be viewed as an alternative, declarative semantics for the planning formalism. Originally, the first version of PDDL [GHK+98] was not accompanied by a formal semantics. The language was rather intended as a standard syntax for a commonly accepted semantics for STRIPS, as described by Lifschitz [Lif87]. The first formal semantics was provided for PDDL version 2.1 by Fox and Long [FL03], who extended Lifschitz’ state-transitional definition to also cope with the newly added numeric and temporal features of PDDL. In any case, the meaning of action operators is defined meta-theoretically through the addition and deletion of literals, which already Lin and Reiter [LR97] identified as problematic, in particular for logically incomplete theories. Moreover, the additional machinery for the temporal and numeric features of PDDL 2.1 makes the semantics rather complicated (in [FL03], the description spans 17 pages) and thus quite difficult to grasp for a human reader.

In [LR97], Lin and Reiter show that STRIPS can be viewed as a mechanism for computing the progression with respect to a certain form of successor state axioms, which they call strongly context free:

\[
\square[a]F(\bar{x}) \equiv (\exists \bar{v}_1) a = g_1(\bar{w}_1) \lor \cdots \lor (\exists \bar{v}_m) a = g_m(\bar{w}_m) \lor F(\bar{x}) \land \neg(\exists \bar{y}_1) a = h_1(\bar{z}_1) \land \cdots \land \neg(\exists \bar{y}_n) a = h_n(\bar{z}_n)
\]

(4.3)

The \( g_i \) and \( h_j \) are action symbols, not necessarily distinct, and the \( \bar{w}_i \) and \( \bar{z}_j \) are variables
that include all of the $\vec{x}$, and where the remaining variables are the existentially quantified $\vec{v}_i$ and $\vec{y}_j$, respectively. The correspondence with STRIPS becomes clear once we instantiate the successor state axioms by some ground action $g(\vec{o})$. Simplifying the resulting formula using the unique names assumption for actions results in a formula of the form:

$\Box[g(\vec{o})]F(\vec{x}) \equiv \vec{x} = \vec{o}_1 \lor \cdots \lor \vec{x} = \vec{o}_m \lor F(\vec{x}) \land \vec{x} \neq \vec{c}_1 \land \cdots \land \vec{x} \neq \vec{c}_n$

(4.4)

In the most common closed-world case, the $\vec{o}_i$ correspond to the instances of $F$ that have to be added, while the $\vec{c}_i$ are the ones that need to be deleted from the current state. In this case, the assumption is that the only situation-dependent sentences in the initial database $\Sigma_0$ are of the form

$F(\vec{x}) \equiv \vec{x} = \vec{d}_1 \lor \cdots \lor \vec{x} = \vec{d}_r$

(4.5)

where we have one such sentence for each fluent $F$. The progression can now easily be determined when we insert (4.5) into (4.4), which yields

$[g(\vec{o})]F(\vec{x}) \equiv \vec{x} = \vec{d}_1 \lor \cdots \lor \vec{x} = \vec{d}_r \\
(\vec{x} = \vec{d}_1 \lor \cdots \lor \vec{x} = \vec{d}_r) \land \vec{x} \neq \vec{c}_1 \land \cdots \land \vec{x} \neq \vec{c}_n.$

(4.6)

This sentence can further be simplified when we assume that the $\vec{o}_i$, $\vec{d}_i$ and $\vec{c}_i$ are all standard names. Because of them being unique, $\vec{x} = \vec{d}_1$ already implies $\vec{x} \neq \vec{c}_j$ for all $\vec{c}_j$ distinct from $\vec{d}_1$. We can therefore drop the inequalities $\vec{x} \neq \vec{c}_j$ and simply remove all the $\vec{x} = \vec{d}_i$ such that $\vec{d}_i$ is identical to any one of the $\vec{c}_j$, which gives a sentence of the form

$[g(\vec{o})]F(\vec{x}) \equiv \vec{x} = \vec{d}_1 \lor \cdots \lor \vec{x} = \vec{d}_q \\
(\vec{x} = \vec{d}_1 \lor \cdots \lor \vec{x} = \vec{d}_q) \land \vec{x} = \vec{d}_{i_1} \lor \cdots \lor \vec{x} = \vec{d}_{i_q}.$

(4.7)

When we do this transformation for all the fluents in the basic action theory, and if we drop the leading $[g(\vec{o})]$ operators, what we obtain is nothing else than the progression of an initial database of sentences of the form (4.5) through $g(\vec{o})$ with respect to the strongly context free successor state axioms (4.3)! Moreover, since (4.7) is of the same form as (4.5), this procedure can be iterated for further actions.

Lin and Reiter have thus shown an exact correspondence between how states and their updates are represented within a STRIPS system, and a certain form of Situation Calculus basic action theories. They argue that the state-transitional semantics for STRIPS defined through meta-theoretic addition and deletion of literals can be viewed as a mere mechanism for computing the progression of a corresponding BAT through a given action. The basic action theory on the other hand is considered the declarative specification of the system, given in the form of axioms that are expressed in a logical language whose semantics is well-defined and
whose properties are well understood. Moreover, obtaining the STRIPS operators from the precondition and successor state axioms is a completely mechanical process, meaning that is straightforward to devise a procedure that compiles the Situation Calculus axiomatization into a corresponding STRIPS description.

It is with this very same motivation that we will in the following provide a Situation-Calculus-based (or more precisely, $\mathcal{ES}$-based) declarative semantics for the Planning Domain Definition Language PDDL, which is basically an extension of STRIPS by numerous new features. As mentioned above, the existing, meta-theoretic definition of PDDL’s semantics is lengthy and complicated. We believe that our alternative, declarative semantics is not only more accessible to a human reader, but that a common, logical representation also allows an easy comparison to other action formalisms. Finally, the new semantics provides the theoretical justification for embedding PDDL-based planners into the GOLOG system, as described above.

It should be noted that resorting to $\mathcal{ES}$ instead of the classical Situation Calculus brings at least two advantages in this context: On the one hand, in many planning formalisms (including PDDL) the assumption is made that distinct constants $c$ and $d$ refer to distinct individuals. To achieve this in the Situation Calculus, we have to include pairwise inequalities ($c \neq d$) in our background axiomatization. Since the distinctness of standard names is already built into $\mathcal{ES}$’s semantics, no such inequality axioms are required when we treat all such $c$ and $d$ as standard names. On the other hand, and more importantly, there are no situation terms in $\mathcal{ES}$. If $\text{Holding}(x)$ is a fluent predicate that is used in the planning formalism, its counterpart in the Situation Calculus requires to be extended by a situation argument, yielding $\text{Holding}(x, s)$. As $\mathcal{ES}$ does not contain situation terms, no such transformation is needed, and indeed entire formulas can be used across both formalisms without modification.

The remainder of this chapter is organized as follows. The following section gives a general introduction to the Planning Domain Definition Language PDDL. Section 4.2 then shows how the semantics of an important subset of PDDL, namely its ADL fragment, can be represented by $\mathcal{ES}$ basic action theories, including a formal proof of the correctness of the mapping. An empirical evaluation as presented in Section 4.3 shows that embedding a PDDL planner into GOLOG is indeed beneficial in terms of the system’s overall runtime, without any loss in expressiveness.

### 4.1 The Planning Domain Definition Language

Since the introduction of STRIPS [FN71] in 1971, AI planning has been an area of ongoing research. While the STRIPS planning system itself was definitely ground-breaking, its rep-
representation formalism was far more influential. Most planners that were developed after the first linear planners of the early 1970s [Sus75], be it the partial-order planners that dominated in the following two decades [PW92] or the seminal graph-planning systems of the mid 1990s [BF97], were based on some variant, extension or derivative of the original STRIPS language. Most notably, Pednault proposed his Action Description Language ADL [Ped89, Ped94] as a “middle-ground” between the highly restricted original STRIPS and the full first-order expressiveness of the Situation Calculus. In particular, the formalism kept the action-centered view of STRIPS, but instead of the simple add and delete lists allowed to attach a condition (in the form of a formula) to each effect, and instead of only allowing positive ground atoms to be tested in action preconditions, allowed to include negation, disjunction, and quantification.

The different planners that have been developed over the years supported different input languages, i.e. extensions of STRIPS or subsets of ADL. The UCPOP system [PW92] for instance could handle planning problems that contain universal quantification and conditional effects (hence the letters 'U' and 'C' in the acronym), but neither existential quantification nor disjunction in preconditions. Furthermore, each system used its own syntax. Direct comparisons between planners were hence done only occasionally, and it was therefore difficult to judge the overall progress in the field.

For this reason, and inspired by similar activities in other fields, the International Planning Competition (IPC) was introduced and first held at the 1998 AI Planning Systems conference (AIPS). The goal of the competition was to create a repository of planning domains and problems, to provide a benchmark for planners that allow researchers to compare their system to the state of the art, and to focus and drive research towards more realistic applications.

The Planning Domain Definition Language PDDL [GHK+98] was designed by the competition committee to serve as the standardized input syntax for the systems participating in the competition. One of the major design goals was that the planning domains and problems formulated with PDDL should only provide “physics, not advice”, i.e. it only describes what the world is like and how actions can possibly change it, but does not give any hints on which actions need to be applied in which order to achieve certain goals.

At the most basic level, PDDL simply provides yet another new representation for the STRIPS formalism. On top of that, the current version 3 of the language includes all of the expressiveness of ADL, an explicit (numeric) notion of time, durative processes, plan constraints and preferences, and more. To be able to identify the different fragments, PDDL uses so-called requirement flags that indicate which features of the language are used in the formulation of a given planning domain. Since typically a planning system only supports a certain subset of the features, it is thus easy for the planner to decide whether it can deal with a given problem or
PDDL utilizes a LISP-like syntax. The general domain of a planning problem is defined separate from the problem itself so that the same domain definition can be reused for different planning problem instances. Figure 4.2 provides an example definition of the office robot domain in PDDL. After the declaration of the domain name, a list of the requirements for the domain is given. In this case, the domain uses a simple STRIPS representation, but where in addition domain objects will be typed. The following lines then defines \texttt{physob} (“physical object”) and \texttt{location} as the types used within the domain, and \texttt{robot} as a constant of type \texttt{physob}, where the term “constant” in this context refers to an object name that has to be present in all instances of this domain. It follows a declaration of the (fluent) predicates, in this case \texttt{at}, \texttt{holding}, \texttt{hand-empty}, and \texttt{portable}, along with their arguments and their corresponding types. The leading question marks are used to distinguish variables as well as predicate and action arguments from other designators.

Next is a list of the action operators, each of which being given by its parameters, its precondition, and its effect. For the \texttt{pickup} action for instance, we require that the object \(?x\) to be picked up is at the same location \(?l\) as the robot, that the robot’s hand is currently empty, and that \(?x\) is a portable object. As an effect, the robot will be holding the object after the action, and its hand will no longer be empty.

The formulation here may seem a bit redundant to the reader. Note that this is due to the \texttt{:strips} requirement flag being set, which implies that preconditions may not contain any negation, disjunction or quantification. This disallows that we test in the precondition whether there \texttt{exists} a location that is the same for the \texttt{robot} and \(?x\), which is why the location was put as an additional argument of the action. For the same reason, we cannot simply check whether the robot’s hand is empty by requiring that there does not \texttt{exist} any object currently being held, and hence the additional predicate \texttt{hand-empty} is used to encode this property. Functional fluents such as the ones we used in the basic action theory of Section 3.3 are not included in the \texttt{:strips} fragment either, and hence the location of physical objects is encoded by the predicate \texttt{at} in this domain formulation.

None of the above would be necessary in the more expressive \texttt{:adl} fragment. Note however that it is quite common that IPC planning domains are reformulated for different fragments of PDDL because of the above mentioned fact that certain planners can handle only certain parts of the language. Yet this does not mean that the higher expressiveness is of no use. On the contrary: While it is most of the time possible to “compile away” certain features, generally the problem description, the length of the plan, or both will considerably grow in size, in the worst case up to an exponential blow-up [Neb00].
(define (domain office-robot)
  (:requirements :strips :typing)
  (:types physob location)
  (:constants robot - physob)
  (:predicates (at ?x - physob ?l - location)
                (holding ?x - physob)
                (hand-empty)
                (portable ?x - physob)
    ...
  )
  (:action pickup
    :parameters (?x - physob ?l - location)
    :precondition (and (at ?x ?l)
                       (at robot ?l)
                       (hand-empty)
                       (portable ?x)
    )
    :effect (and (holding ?x)
                 (not (hand-empty)))
  )
  (:action putdown
    ...
  )
)

Figure 4.2: Office robot domain defined in the STRIPS fragment of PDDL
(define (problem office-robot-problem)
  (:domain office-robot)
  (:objects cup1 cup2 - physob
    office-ann office-bob kitchen hallway - location)
  (:init (at robot hallway)
    (at cup1 kitchen)
    (at cup2 kitchen)
    (hand-empty)
    (portable cup1)
    (portable cup2)
  )
  (:goal (and (at cup1 office-ann) (at cup2 office-bob)))
)

Figure 4.3: PDDL planning problem instance for the office robot domain

Figure 4.3 shows an exemplary problem instance for the above domain. In the first line
the problem is given a name, and in the second one the corresponding planning domain is
referenced. It follows a list of the objects in the domain with associated types, where in this
case the notation is to be read such that both cup1 and cup2 are of type physob, whereas
office-ann, office-bob, kitchen and hallway are all of type location. Next, the initial
state of the planning problem is provided in the form of positive literals (atoms) that are
supposed to hold initially. In this case, the robot initially is at the hallway, whereas cup1 and
cup2 are in the kitchen, the robot’s hand is empty, and both cups are portable objects. Note that
PDDL includes the closed-world assumption: Every fact that is not explicitly mentioned to be
true here is assumed to be false. For example, since nothing is said within the initial description
about the state of the predicate holding, the atoms (holding cup1) and (holding cup2) will
by default not hold initially. Finally, a goal is provided in the form of a formula for which the
same restrictions apply as for preconditions of actions. As in this domain only the :strips
requirement flag is set, the formula has to be a conjunction of positive literals.

The IPC has taken place biennially since 1998.\footnote{More precisely, up to today, the competition was held in the years 1998, 2000, 2002, 2004, 2006, 2008, and 2011.} The PDDL language has undergone several
revisions since then, where new features were introduced, but also others were dropped. The first
version for the 1998 competition, presented in [GHK+98], supported on top of basic STRIPS-style action definitions also the conditional effects of ADL. Moreover, it included definitions for axioms, safety constraints, hierarchical actions, and numerical fluents. Here, an “axiom” means a formula that relates the truth value of a certain predicate that is not explicitly defined to the values of other predicates. Safety constraints are formulas that invalidate a plan once they get violated at some intermediate state. The term “hierarchical actions” refers to the possibility to decompose actions into several subtasks as it is done in hierarchical task network planning. Numerical fluents are state-dependent functions whose arguments are from the set of domain objects and whose values are numbers. Furthermore, in addition to the standard closed-world case, the original PDDL also allowed to define open-world planning problems where states were represented as consistent, but not necessarily complete sets of literals. If such a state contains a positive literal \( P \), then the atom \( P \) is known to be true in that state, whereas if \( \neg P \) is contained, \( P \) is known to be false. If \( P \) is not mentioned at all, then the truth state of \( P \) is unknown.

None of these additional features were actually used in the 1998 planning competition, as all five of the participating planners supported either STRIPS only or at most the ADL subset. Similarly, only the (closed-world) ADL fragment was used in the 2000 competition. Every subsequent extension of PDDL afterwards then added to this subset, rather than the original PDDL.

The first such extension was for the third IPC in 2002. In [FL03], Fox and Long present version 2.1 of PDDL that extends the previous version by numerics, time, and concurrent durative processes. They propose a hierarchy of sublanguages of increasing expressiveness:

- Level 1 consists of the purely logical, non-numerical STRIPS and ADL subsets with instantaneous actions;
- level 2 contains all of level 1, and additionally allows the usage of numeric fluent functions;
- level 3 extends level 2 by discretized durative actions;
- level 4 in addition includes continuous durative actions and
- level 5 contains all of the above plus spontaneous events and physical processes.

The difference between discretized and continuous durative actions is that the former have only effects that take place at the start or end *time points* of the action’s durations, whereas the latter may also contain effects that induce a continuous change of some numeric property during the *time interval* in which the action is executed. Level 5 furthermore is actually not part of the “official” PDDL 2.1, but an extension called PDDL+ [FL06] which introduces
autonomous processes that may be either triggered by the agent’s actions itself or by effects of other processes.

The next update was provided with PDDL 2.2 [EH04], which was used at the fourth competition in 2004. It adds two new features to the previous version: On the one hand, so-called timed-initial literals can be provided in the description of the initial state. A ground literal can thus be defined to become true at a pre-determined, explicit time point, which is mainly useful to encode time windows in which certain activities have to or must not occur, e.g. the opening hours of shops. On the other hand, version 2.2 re-introduces the axioms of PDDL 1.2, where the corresponding fluents that are defined by those axioms are now referred to as derived predicates. There is a good reason for the latter: As it was shown in [THN05], although it is in general possible to “compile away” such axioms, such a compilation cannot be done if it is required that the size of planning problem description and the length of the plan do not grow more than polynomially. This is to say that derived predicates are not only “syntactic sugar”, but they contribute a reasonable amount of expressiveness to the formalism.

For the fourth IPC in 2006, PDDL was yet extended again, yielding version 3.0. It allows to define trajectory constraints that express requirements about the intermediate steps of a plan, as opposed to normal goals which only talk about the final state to be reached. Moreover, PDDL 3.0 provides the possibility to express soft goals and constraints, also called preferences. As opposed to their strong counterparts, soft constraints are not required to hold for a plan to be valid, but are considered “nice to have”, where the more of them are fulfilled in the plan, the more preferred it is. Preferences in PDDL are defined quantitatively, i.e. plan quality is measured by a numeric metric, whose value depends on which soft goals are satisfied (or not) in the given plan.

Figure 4.4 presents an overview of all requirement flags supported by PDDL 3.0, along with their corresponding meaning.

4.2 The ADL Fragment

As discussed earlier, Lin and Reiter already showed in [LR97] how the STRIPS updating mechanism of adding and deleting literals corresponds to progression with respect to strongly context-free successor state axioms. In this section, a similar mapping is provided for PDDL’s ADL subset, which is obtained by using that part of the PDDL syntax described in [GL05a] that is accessible through the :adl requirement flag. As can be seen from Figure 4.4, this means that this sublanguage extends STRIPS by supporting equality, conditional effects, typing, as well as negation, disjunction and quantifiers within preconditions. The main result of this
### Requirement Description

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:strips</td>
<td>Basic STRIPS-style adds and deletes.</td>
</tr>
<tr>
<td>:typing</td>
<td>Allow type names in declarations of variables.</td>
</tr>
<tr>
<td>:negative-preconditions</td>
<td>Allow not in goal descriptions.</td>
</tr>
<tr>
<td>:disjunctive-preconditions</td>
<td>Allow or in goal descriptions.</td>
</tr>
<tr>
<td>:equality</td>
<td>Support = as built-in predicate.</td>
</tr>
<tr>
<td>:existential-preconditions</td>
<td>Allow exists in goal descriptions.</td>
</tr>
<tr>
<td>:universal-preconditions</td>
<td>Allow forall in goal descriptions.</td>
</tr>
<tr>
<td>:quantified-preconditions</td>
<td>= :existential-preconditions + :universal-preconditions</td>
</tr>
<tr>
<td>:conditional-effects</td>
<td>Allow when in action effects.</td>
</tr>
<tr>
<td>:fluents</td>
<td>Allow function definitions and use of effects using assign-</td>
</tr>
<tr>
<td></td>
<td>ment operators and arithmetic preconditions.</td>
</tr>
<tr>
<td>:adl</td>
<td>= :strips + :typing + :negative-preconditions +</td>
</tr>
<tr>
<td></td>
<td>:disjunctive-preconditions + :equality +</td>
</tr>
<tr>
<td></td>
<td>:quantified-preconditions + :conditional-effects</td>
</tr>
<tr>
<td>:durative-actions</td>
<td>Allows durative actions. Note that this does not imply :fluents.</td>
</tr>
<tr>
<td>:derived-predicates</td>
<td>Allows predicates whose truth value is defined by a for-</td>
</tr>
<tr>
<td></td>
<td>mula.</td>
</tr>
<tr>
<td>:timed-initial-literals</td>
<td>Allows the initial state to specify literals that will be-</td>
</tr>
<tr>
<td></td>
<td>come true at a specified time point, implies :durative-a-</td>
</tr>
<tr>
<td></td>
<td>ctions.</td>
</tr>
<tr>
<td>:preferences</td>
<td>Allows use of preferences in action preconditions and go-</td>
</tr>
<tr>
<td></td>
<td>als.</td>
</tr>
<tr>
<td>:constraints</td>
<td>Allows use of constraints fields in domain and problem f-</td>
</tr>
<tr>
<td></td>
<td>iles. These may contain modal operators supporting trajec-</td>
</tr>
<tr>
<td></td>
<td>tory constraints.</td>
</tr>
</tbody>
</table>

Figure 4.4: PDDL requirement flags and their meaning. Source: [GL05a]
section can thus be seen as a generalization of Lin and Reiter’s.

We start by presenting the language’s syntax and semantics, after which the mapping to $\mathcal{ES}$ basic action theories is shown, followed by a corresponding correctness proof and a discussion of how the two formalisms relate in terms of expressiveness.

### 4.2.1 Language Definition

Instead of resorting to the original LISP-like syntax of PDDL, we use a more logic-like notation here. In particular, we will already use symbols from the $\mathcal{ES}$ alphabet to define ADL problems, which simplifies the later mapping between the two formalisms. We need the following formal definitions:

**Definition 4.1 (ADL Domains).** An ADL domain is given by $D = \langle T, E, O_D, F, A \rangle$, where

- $T = \{ \tau_1, \ldots, \tau_k, Object \}$ is a finite list of types, each of which is a unary rigid predicate symbol, and where Object is a special type that is always included;

- $E$ is a finite list of statements of the form
  \[
  \tau_i : (\text{either } \tau_{i1} \cdots \tau_{ik_i}),
  \]
  where $k_i \geq 2$, and which intuitively means that $\tau_i \in T \setminus \{Object\}$ is a compound type defined as the union of the distinct $\tau_{ij} \in T \setminus \{Object\}$. We require that each type appears on the left-hand side of at most one such statement, and that if $\tau_{ij}$ appears on the right-hand side of an “either” statement, it does not appear on the left-hand side of any. Types distinct from Object that are not defined as compound types in this way are called primitive. In the special case that the number $k$ of types distinct from Object in $T$ is zero, we consider Object a primitive type. Otherwise, it is implicitly understood as the most general supertype, i.e. the union of all primitive types.

- $O_D = \{ o_1:\tau_o_1, \ldots, o_l:\tau_o_l \}$ is a finite list of “constants” with associated types, where each $o_i$ is an object standard name and each $\tau_o_i \in T$ is primitive;

- $F = \{ F_1: (\tau_{F_1}^1 \cdots \tau_{F_1}^{m_1}), \ldots, F_m: (\tau_{F_m}^1 \cdots \tau_{F_m}^{m_m}) \}$ is a finite list of fluent predicates with associated types (from $T$) for their arguments;

- and $A = \{ \omega_1, \ldots, \omega_n \}$ is a finite list of action operators with respect to $\langle T, E, O_D, F \rangle$ according to Definition 4.4.

Here and in the following we also assume without further mentioning that symbols such as constants, predicates and the like are always distinct.
Definition 4.2 (ADL Problems). An ADL problem is given by $P = \langle D, O_p, I, G \rangle$, where

- $D = \langle T, E, O_b, F, A \rangle$ is an ADL domain according to Definition 4.1;
- $O_p = \{ o_{t+1}:r_{o_{t+1}}, \ldots, o_{l+q}:r_{o_{l+q}} \}$ is a finite list of object standard names with associated primitive types from $T$,
- $I$ is the initial state in form of a finite set of ground atoms with respect to $\langle T, E, O_b \cup O_p, F \rangle$ according to Definition 4.3;
- $G$ is the goal in form of a precondition formula with respect to $\langle T, E, O_b \cup O_p, F, \emptyset \rangle$ according to Definition 4.5.

Definition 4.3 (ADL Atoms). An ADL atom with respect to $\langle T, E, O, F, V \rangle$ is of the form $F_i(o_{1}, \ldots, o_{m_i})$, where $F_i(\tau_{F_{1}}, \ldots, \tau_{F_{l_i}}) \in F$ and for each $t_j$, either $t_j$ is an object variable such that $(t_j; \tau) \in V$ and $\tau \leq \tau_{F_j}$, or $t_j$ is an object standard name with $(t_j; \tau) \in O$ and $\tau \leq \tau_{F_j}$. Here, by $\tau \leq \tau'$ we mean that either $\tau$ and $\tau'$ are identical, or $(\tau'; (\tau \cdots \tau \cdots)) \in E$, or $\tau'$ is identical to Object. An ADL ground atom with respect to $\langle T, E, O, F \rangle$ is an ADL atom without variables, i.e. an ADL atom with respect to $\langle T, E, O, F, \emptyset \rangle$.

Definition 4.4 (ADL Operators). An ADL operator with respect to $\langle T, E, O_b, F \rangle$ is given by $\omega_j = \langle g_j, \varpi_j, \pi_j, \epsilon_j \rangle$, where

- $g_j$ is an $n_j$-ary action function symbol;
- $\varpi_j = \{ y_{g_j}^1 : r_{g_j}^1, \ldots, y_{g_j}^{n_j} : r_{g_j}^{n_j} \}$ is a list of object variables with associated types, called the parameters of $\omega_j$;
- $\pi_j$ is a precondition formula with respect to $\langle T, E, O_b, F, \varpi_j \rangle$ according to Definition 4.5;
- $\epsilon_j$ is an ADL effect with respect to $\langle T, E, O_b, F, g_j, \varpi_j \rangle$ according to Definition 4.6.

Definition 4.5 (ADL Precondition Formulas). The ADL precondition formulas with respect to $\langle T, E, O, F, V \rangle$ are the least set such that:

1. If $F(t)$ is an ADL atom with respect to $\langle T, E, O, F, V \rangle$, then it is an ADL precondition formula with respect to $\langle T, E, O, F, V \rangle$.
2. If for each $t_i$, either $t_i$ is an object variable such that $(t_i; \tau) \in V$ or $t_i$ is an object standard name with $(t_i; \tau) \in O$, then $(t_1 = t_2)$ is an ADL precondition formula with respect to $\langle T, E, O, F, V \rangle$. Note that as opposed to the case above, no type consistency is required here.
3. If $\phi_1$ and $\phi_2$ are ADL precondition formulas with respect to $\langle T, E, O, F, V \rangle$, then $\phi_1 \land \phi_2$ and $\neg \phi_1$ are also ADL precondition formulas with respect to $\langle T, E, O, F, V \rangle$.

4. If $\phi$ is an ADL precondition formula with respect to $\langle T, E, O, F, V \rangle$ and $(x:t) \in V$ is an object variable with associated type $t \in T$, then $\forall x:t. \phi$ is an ADL precondition formula with respect to $\langle T, E, O, F, V \backslash \{x:t\} \rangle$.

Definition 4.6 (ADL Effects). An ADL effect with respect to $\langle T, E, O_D, F, g_j, \wp_j \rangle$ is a finite collection of expressions of the form

\begin{equation}
\forall \vec{x}_i: \vec{\tau}_{F_i}. \gamma_{F_i, g_j}^+ \Rightarrow F_i(\vec{x}_i), \tag{4.9}
\end{equation}

and expressions of the form

\begin{equation}
\forall \vec{x}_i: \vec{\tau}_{F_i}. \vec{\gamma}_{F_i, g_j}^- \Rightarrow \neg F_i(\vec{x}_i), \tag{4.10}
\end{equation}

where $(\vec{F}_i; \vec{\tau}_{F_i}) \in F$, and the effect conditions $\gamma_{F_i, g_j}^+$ and $\vec{\gamma}_{F_i, g_j}^-$ are precondition formulas with respect to $\langle T, E, O_D, F, \{\vec{x}_i: \vec{\tau}_{F_i}\} \cup \wp_j \rangle$. We call (4.9) a positive effect and (4.10) a negative effect for $F_i$.

Example 4.7. As an example, consider our office robot again. One possible encoding is given as follows. Let the domain be

$$D = \langle T, E, O_D, F, A \rangle$$

where

$$T = \{\text{Physob, Location, Robot, Cup, Letter, Object}\}$$

$$E = \{\text{Physob: (either Robot Cup Letter)}\}$$

$$O_D = \{\text{robot: Robot}\}$$

$$F = \{\text{At: (Physob Location), Holding: (Physob), Portable: (Physob), Nextto: (Location Location)}\}$$

$$A = \{\omega_1, \omega_2, \omega_3\}$$

The operators are defined as follows:

$$\omega_1 = \langle \text{pickup},$$

$$\{y: \text{Physob}\},$$

$$\exists x: \text{Location. } \text{At} (\text{robot}, x) \land \text{At} (y, x) \land \text{Portable} (y) \land \neg \exists x' \text{Holding} (x'),$$

$$\{\forall x: \text{Physob. } x = y \Rightarrow \text{Holding} (x) \} \rangle$$
4.2 The ADL Fragment

$$\omega_2 = \langle \text{putdown}, \{ y : \text{Physob} \}, \text{Holding}(y), \{ \forall x : \text{Physob}. x = y \Rightarrow \neg \text{Holding}(x) \} \rangle$$

$$\omega_3 = \langle \text{goto}, \{ y_1 : \text{Location}, y_2 : \text{Location} \}, \text{At}(\text{robot}, y_1) \land (\text{NextTo}(y_1, y_2) \lor \text{NextTo}(y_2, y_1)), \{ \forall x_1 : \text{Physob} \forall x_2 : \text{Location}. (x_1 = \text{robot} \land x_2 = y_2) \Rightarrow \text{At}(x_1, x_2), \forall x_1 : \text{Physob} \forall x_2 : \text{Location}. (x_1 = \text{robot} \land x_2 = y_1) \Rightarrow \neg \text{At}(x_1, x_2), \forall x_1 : \text{Physob} \forall x_2 : \text{Location}. (\text{Holding}(x_1) \land x_2 = y_2) \Rightarrow \text{At}(x_1, x_2), \forall x_1 : \text{Physob} \forall x_2 : \text{Location}. (\text{Holding}(x_1) \land x_2 = y_1) \Rightarrow \neg \text{At}(x_1, x_2) \} \rangle$$

Note that as opposed to the formalization given in Figure 4.2 which uses only the STRIPS subset of PDDL, we here exploit the full expressiveness of the ADL fragment. In particular, preconditions such as $$\neg \exists x' \text{Holding}(x')$$ may contain negation and quantification, which is why additional predicates like HandEmpty are unnecessary. Moreover, conditional effects allow us to properly encode the fact that when the robot moves, all objects that it holds move along with it. An example problem for this domain is

$$P = \langle D, O_P, I, G \rangle$$

where

$$O_P = \{ \text{cup}_1 : \text{Cup}, \text{cup}_2 : \text{Cup}, \text{officeann} : \text{Location}, \text{officebob} : \text{Location}, \text{kitchen} : \text{Location}, \text{hallway} : \text{Location} \}$$

$$I = \{ \text{At}(\text{robot}, \text{hallway}), \text{At}(\text{cup}_1, \text{hallway}), \text{At}(\text{cup}_2, \text{kitchen}), \text{Holding(\text{cup}_1}), \text{Portable(\text{cup}_1)}, \text{Portable(\text{cup}_2)}, \text{NextTo(officeann, hallway)}, \text{NextTo(officebob, hallway)}, \text{NextTo(kitchen, hallway)} \}$$

$$G = \text{At(\text{cup}_1, \text{officeann})} \land \text{At(\text{cup}_2, \text{officebob})}$$

### 4.2.2 Semantics

**Definition 4.8 (ADL Object Domains).** Let $$\tau \in T$$ be a type. We define $$|\tau|(T, E, O)$$, the *domain of $$\tau$$ with respect to $$\langle T, E, O \rangle$$*, inductively as follows:

- $$|\tau|(T, E, O) = \{ o \mid (o: \tau) \in O \}$$, if $$\tau \in T$$ is primitive;
\( |\tau|(T,E,0) = |\tau_i|(T,E,0) \cup \cdots \cup |\tau_{k_i}|(T,E,0) \), if \( \tau : (\text{either } \tau_1 \cdots \tau_{k_i}) \in E \);

\( |\tau|(T,E,0) = \{ o \mid (o;\tau') \in 0 \} \), if \( \tau = \text{Object} \).

**Definition 4.9 (ADL States).** An ADL state with respect to \( \langle T, E, 0, F \rangle \) is a finite set of ground atoms with respect to \( \langle T, E, 0, F \rangle \).

**Definition 4.10 (Truth of ADL Formulas).** Let \( S \) be an ADL state with respect to \( \langle T, E, 0, F \rangle \). Then

1. \( S \models F(\vec{o}) \) iff \( F(\vec{o}) \in S \);
2. \( S \models (o_1 = o_2) \) iff \( o_1 \) and \( o_2 \) are identical;
3. \( S \models \phi_1 \land \phi_2 \) iff \( S \models \phi_1 \) and \( S \models \phi_2 \);
4. \( S \models \neg \phi \) iff \( S \not\models \phi \);
5. \( S \models \forall x : \tau. \phi \) iff \( S \models \bigwedge_{o \in |\tau|(T,E,0)} \phi_o^x \).

**Definition 4.11 (ADL Ground Actions).** Let \( \omega_j = (g_j, \vec{y}_j; \vec{\tau}_j, \pi_j, \epsilon_j) \) be an ADL operator with respect to \( \langle T, E, 0_0, F \rangle \). A ground action for \( \omega_j \) with respect to \( 0 \) is a term of the form \( g_j(\vec{o}) \), where \( \vec{o} \in |\vec{\tau}_j|(T,E,0) \).

**Definition 4.12 (ADL Updates).** Let \( S \) be an ADL state with respect to \( \langle T, E, 0, F \rangle \) and \( g_j(\vec{o}) \) a ground action for \( \omega_j = (g_j, \vec{y}_j; \vec{\tau}_j, \pi_j, \epsilon_j) \) with respect to \( 0 \). Then

\[
\text{(4.11)} \quad \text{Adds} = \{ F_i(\vec{c}) \mid (\forall \vec{x}_i : \vec{\tau}_F, \gamma_{F_i,g_j}^+ \Rightarrow F_i(\vec{x}_i)) \in \epsilon_j, \quad \vec{c} \in |\vec{\tau}_F|(T,E,0) \text{ and } S \models (\gamma_{F_i,g_j}^+)^{\vec{x}_i}_{\vec{y}_i} \}
\]

\[
\text{(4.12)} \quad \text{Dels} = \{ F_i(\vec{c}) \mid (\forall \vec{x}_i : \vec{\tau}_F, \gamma_{F_i,g_j}^- \Rightarrow \neg F_i(\vec{x}_i)) \in \epsilon_j, \quad \vec{c} \in |\vec{\tau}_F|(T,E,0) \text{ and } S \models (\gamma_{F_i,g_j}^-)^{\vec{x}_i}_{\vec{y}_i} \}
\]

The updated state for \( S \) with respect to \( g_j(\vec{o}) \) is

\[
\text{(4.13)} \quad S' = (S \setminus \text{Dels}) \cup \text{Adds}.
\]

**Definition 4.13 (ADL Plans).** Let \( P = \langle D, 0_P, I, G \rangle \) be an ADL problem and \( D = \langle T, E, 0_D, F, A \rangle \) a corresponding ADL domain. A plan \( P \) for \( P \) is given by a finite sequence of ground actions \( \langle g_{j_1}(\vec{o}_1), \ldots, g_{j_k}(\vec{o}_k) \rangle \) for the operators \( \omega_j = (g_j, \vec{y}_j; \pi_j, \epsilon_j) \in A \) with respect to \( 0_D \cup 0_P \). \( P \) induces a trace of states \( S^0, S^1, \ldots, S^k \), where \( S^0 = I \) is the initial state, and each \( S^i \) is the updated
4.2 The ADL Fragment

state for $S_{i-1}$ with respect to $g_{ji}(\overline{o})$. The plan is executable when for every $1 \leq i \leq k$, the precondition is satisfied, i.e.

\[(4.14) \quad S^i \models (\pi_{ji})_{\overline{o}_i}.\]

It is furthermore valid if it is executable and the goal formula holds in the final state:

\[(4.15) \quad S^k \models G.\]

**Example 4.14.** Let $S^0$ be the state that is given by the initial state $I$ given in Example 4.7:

$$S^0 = \{ \text{At}(\text{robot}, \text{hallway}), \text{At}(\text{cup}_1, \text{hallway}), \text{At}(\text{cup}_2, \text{kitchen}),$$
$$\text{Holding}(\text{cup}_1), \text{Portable}(\text{cup}_1), \text{Portable}(\text{cup}_2),$$
$$\text{NextTo}(\text{officeann}, \text{hallway}), \text{NextTo}(\text{officebob}, \text{hallway}), \text{NextTo}(\text{kitchen}, \text{hallway}) \}$$

For the ground action $\text{goto}(\text{hallway}, \text{officeann})$, we obtain

$$\text{Adds} = \{ \text{At}(\text{robot}, \text{officeann}), \text{At}(\text{cup}_1, \text{officeann}) \}$$
$$\text{Dels} = \{ \text{At}(\text{robot}, \text{hallway}), \text{At}(\text{cup}_1, \text{hallway}) \}.$$

The updated state for $S^0$ with respect to $\text{goto}(\text{hallway}, \text{officeann})$ thus is

$$S^1 = \{ \text{At}(\text{robot}, \text{officeann}), \text{At}(\text{cup}_1, \text{officeann}), \text{At}(\text{cup}_2, \text{kitchen}),$$
$$\text{Portable}(\text{cup}_1), \text{Portable}(\text{cup}_2),$$
$$\text{NextTo}(\text{officeann}, \text{hallway}), \text{NextTo}(\text{officebob}, \text{hallway}), \text{NextTo}(\text{kitchen}, \text{hallway}) \}.$$

Moreover, it is easy to check that the plan

$$\mathcal{P} = \langle \text{goto}(\text{hallway}, \text{officeann}), \text{putdown}(\text{cup}_1), \text{goto}(\text{officeann}, \text{hallway}), \text{goto}(\text{hallway}, \text{kitchen}),$$
$$\text{pickup}(\text{cup}_2), \text{goto}(\text{kitchen}, \text{hallway}), \text{goto}(\text{hallway}, \text{officebob}), \text{putdown}(\text{cup}_2) \rangle$$

is both executable and valid for the goal formula

$$G = \text{At}(\text{cup}_1, \text{officeann}) \land \text{At}(\text{cup}_2, \text{officebob}).$$

### 4.2.3 The Mapping

Let $\mathcal{P} = (\mathcal{D}, \mathcal{O}_p, I, G)$ be an ADL problem, where $\mathcal{D} = (T, E, O_D, F, A)$ is an ADL domain. As we will see, constructing a corresponding basic action theory $\text{Map}(\mathcal{P})$ is actually quite straightforward. We define

\[(4.16) \quad \text{Map}(\mathcal{P}) = \text{Map}_0(T, E, O_D \cup O_p, F, I) \cup \text{Map}_\text{pre}(A) \cup \text{Map}_\text{post}(F, A),\]

each of which we will now discuss in further detail.
The Initial Database

$Map_0(T,E,O,F,I)$ is the initial database of the translated basic action theory. Obviously, we need it to encode the initial state $I$ of our PDDL problem, which we achieve by requiring that it includes for each $F_i: \vec{r}_{F_i} \in F$ a sentence of the form

$$F_i(\vec{x}_i) \equiv (\vec{x}_i = \vec{o}_1 \lor \cdots \lor \vec{x}_i = \vec{o}_{k_o}), \tag{4.17}$$

where $\{\vec{o}_1, \ldots, \vec{o}_{k_o}\} = \{\vec{o} \mid F_i(\vec{o}) \in I\}$.

Additionally, we make $Map_0(T,E,O,F,I)$ contain all information about typing. For each primitive type $\tau_i \in T$, we therefore include a sentence of the form

$$\tau_i(x) \equiv (x = o_{i_1} \lor \cdots \lor x = o_{i_{i_i}}), \tag{4.18}$$

where $\{o_{i_1}, \ldots, o_{i_{i_i}}\} = \{o \mid (o:\tau_i) \in O\}$. Furthermore, for each compound type definition $(\tau_i: (\text{either} \tau_{i_1} \cdots \tau_{i_{k_{\tau_i}}})) \in E$, we have a sentence of the form

$$\tau_i(x) \equiv (\tau_{i_1}(x) \lor \cdots \lor \tau_{i_{k_{\tau_i}}}(x)). \tag{4.19}$$

Finally, in case $Object$ is not primitive, the set also includes the single sentence

$$Object(x) \equiv (\tau_{j_1}(x) \lor \cdots \lor \tau_{j_{j_p}}(x)), \tag{4.20}$$

where $\{\tau_{j_1}, \ldots, \tau_{j_{j_p}}\} = \{\tau \mid \tau \in T \text{ primitive}\}$.

**Example 4.15.** For the domain and problem given in Example 4.7, we obtain:

\[
\begin{align*}
At(x_1,x_2) & \equiv (x_1 = \text{robot} \land x_2 = \text{hallway} \lor x_1 = \text{cup}_1 \land x_2 = \text{hallway} \lor \\
& \quad x_1 = \text{cup}_2 \land x_2 = \text{kitchen}) \\
Holding(x_1) & \equiv (x_1 = \text{cup}_1) \\
Portable(x_1) & \equiv (x_1 = \text{cup}_1 \lor x_1 = \text{cup}_2) \\
Nextto(x_1,x_2) & \equiv (x_1 = \text{officeann} \land x_2 = \text{hallway} \lor x_1 = \text{officebob} \land x_2 = \text{hallway} \lor \\
& \quad x_1 = \text{kitchen} \land x_2 = \text{hallway}) \\
Robot(x) & \equiv (x = \text{robot}) \\
Location(x) & \equiv (x = \text{officeann} \lor x = \text{officebob} \lor x = \text{kitchen} \lor x = \text{hallway}) \\
Cup(x) & \equiv (x = \text{cup}_1 \lor x = \text{cup}_2) \\
Letter(x) & \equiv \perp \\
Physob(x) & \equiv (\text{Robot}(x) \lor \text{Cup}(x) \lor \text{Letter}(x)) \\
Object(x) & \equiv (\text{Location}(x) \lor \text{Robot}(x) \lor \text{Cup}(x) \lor \text{Letter}(x))
\end{align*}
\]
4.2 The ADL Fragment

The Precondition Axiom

In order to construct a precondition axiom, we basically use the simple solution to the qualification problem as discussed in Section 3.3.1. That is to say that for the precondition formula $\pi_j$ of each of the operators $\omega_j = \langle g_j, \pi_j, \pi_j, \epsilon_j \rangle \in A$ we assume a necessary precondition axiom:

\[ \Box \text{Poss}(a) \supset \exists \vec{y}_j : \tau_{g_j}. \; a = g_j(\vec{y}_j) \land \pi_j \]  

(4.21)

Next, we make the completeness assumption that these axioms represent all the necessary and sufficient conditions for an action to be executable, yielding

\[ \pi \overset{def}{=} \bigvee_{1 \leq j \leq n} \exists \vec{y}_j : \tau_{g_j}. \; a = g_j(\vec{y}_j) \land \pi_j. \]  

(4.22)

The single precondition axiom of our basic action theory then is

\[ \text{Map}_{\text{pre}}(A) = \{ \Box \text{Poss}(a) \equiv \pi \}. \]  

(4.23)

Example 4.16. For the domain and problem given in Example 4.7, we obtain:

\[ \Box \text{Poss}(a) = \]  

\[ \exists y : \text{Physob} (a = \text{pickup}(y) \land \exists x : \text{Location}. \; \text{At(robot, } x) \land \text{At}(y, x) \land \text{Portable}(y) \land \neg \exists x' \text{Holding}(x')) \lor \]  

\[ \exists y : \text{Physob} (a = \text{putdown}(y) \land \text{Holding}(y)) \lor \]  

\[ \exists y_1 : \text{Location} \exists y_2 : \text{Location} (\text{At(robot, } y_1) \land (\text{Nextto}(y_1, y_2) \lor \text{Nextto}(y_2, y_1))). \]

The Successor State Axioms

Similarly, the successor state axioms are almost identically constructed as in Reiter’s solution to the frame problem (cf. Section 3.3.2). First, for each fluent predicate $F_i : \tau_{F_i} \in F$, we collect all its positive and negative effects from all operators in $A$:

\[ \gamma_{F_i}^+ \overset{def}{=} \bigvee_{(\forall \vec{x}_i : \tau_{F_i} \cdot \gamma_{F_i, g_j} \Rightarrow F_i(\vec{x}_i)) \in \epsilon_j} \exists \vec{y}_j : \tau_{g_j}. \; a = g_j(\vec{y}_j) \land \gamma_{F_i, g_j}^+ \]  

(4.24)

\[ \gamma_{F_i}^- \overset{def}{=} \bigvee_{(\forall \vec{x}_i : \tau_{F_i} \cdot \gamma_{F_i, g_j} \Rightarrow \neg F_i(\vec{x}_i)) \in \epsilon_j} \exists \vec{y}_j : \tau_{g_j}. \; a = g_j(\vec{y}_j) \land \gamma_{F_i, g_j}^- \]  

(4.25)

The set $\text{Map}_{\text{post}}(F, A)$ then consists of one successor state axiom of the form

\[ \Box [a] F_i(\vec{x}_i) \equiv \gamma_{F_i}^+ \land \tau_{F_i}(\vec{x}_i) \lor F_i(\vec{x}_i) \land \neg \gamma_{F_i}^- \]  

(4.26)
for each of the $F_i$. Note that the only difference to Reiter’s SSAs is that we have the additional requirement that $F_i$’s type constraints have to be met by the $\vec{x}_i$ in order for $F_i(\vec{x}_i)$ to become true.

Example 4.17. For the domain and problem given in Example 4.7, we obtain:

$$
\gamma^+_{At} = \exists y_1 : \text{Location} \exists y_2 : \text{Location} \ (a = \text{goto}(y_1, y_2) \land x_1 = \text{robot} \land x_2 = y_2) \lor \\
\exists y_1 : \text{Location} \exists y_2 : \text{Location} \ (a = \text{goto}(y_1, y_2) \land \text{Holding}(x_1) \land x_2 = y_2)
$$

$$
\gamma^-_{At} = \exists y_1 : \text{Location} \exists y_2 : \text{Location} \ (a = \text{goto}(y_1, y_2) \land x_1 = \text{robot} \land x_2 = y_1) \lor \\
\exists y_1 : \text{Location} \exists y_2 : \text{Location} \ (a = \text{goto}(y_1, y_2) \land \text{Holding}(x_1) \land x_2 = y_1)
$$

$$
\gamma^+_{\text{Holding}} = \exists y : \text{Physob} \ (a = \text{pickup}(y) \land x = y)
$$

$$
\gamma^-_{\text{Holding}} = \exists y : \text{Physob} \ (a = \text{putdown}(y) \land x = y)
$$

$$
\gamma^+_{\text{Portable}} = \gamma^-_{\text{Portable}} = \gamma^+_{\text{Nextto}} = \gamma^-_{\text{Nextto}} = \bot
$$

With simplification, the successor state axioms thus are:

$$
\Box [a] \text{At}(x_1, x_2) \equiv \\
\exists y_1 : \text{Location} \exists y_2 : \text{Location} \ (a = \text{goto}(y_1, y_2) \land \\
(\neg \exists y : \text{Physob}(a = \text{pickup}(y) \land x = y) \lor \\
\neg \exists y : \text{Physob}(a = \text{putdown}(y) \land x = y)))
$$

$$
\Box [a] \text{Holding}(x_1) \equiv \\
\exists y : \text{Physob}(a = \text{pickup}(y) \land x = y) \lor \\
\text{Holding}(x_1) \land \neg (\exists y : \text{Physob}(a = \text{putdown}(y) \land x = y))
$$

$$
\Box [a] \text{Portable}(x_1) \equiv \text{Portable}(x_1)
$$

$$
\Box [a] \text{Nextto}(x_1, x_2) \equiv \text{Nextto}(x_1, x_2)
$$

4.2.4 Correctness

The first observation is that the above constructed theory is indeed of the right form to be a basic action theory:

Lemma 4.18. Let $P$ be an ADL problem. Then $\operatorname{Map}(P)$ is a basic action theory with respect to $\langle D, F \rangle = \langle \{\text{Poss}\}, \{F_1, \ldots, F_n\} \rangle$.

The next lemma shows that $\operatorname{Map}_0$ correctly captures the ADL problem’s typing information:
Lemma 4.19. Let \( S \) be an ADL state with respect to \( \langle T, E, O, F \rangle \), \( \tau \in T \) be a type, \( n \) be any standard name, and \( w \) any world such that \( w \models \text{Map}_0(T, E, O, F, S) \). Then

\[
n \in |\tau|(T, E, O) \iff w \models \tau(n)
\]

Proof. We distinguish the three cases of Definition 4.8:

- \( \tau = \tau_i \in T \) is primitive:

  \[
n \in |\tau|(T, E, O) \\
  \text{iff } (n: \tau) \in O \\
  \text{iff } n \text{ is among the } o_{i_j} \text{ in (4.18)} \\
  \text{iff } w \models \tau(n) \quad \text{(since } w \models (4.18))
\]

- \( \tau = \tau_i \) is compound with \( (\tau_i; (\text{either } \tau_{i_1} \cdots \tau_{i_k})) \in E \):

  \[
n \in |\tau|(T, E, O) \\
  \text{iff } n \in |\tau_{i_1}|(T, E, O) \text{ or } \cdots \text{ or } n \in |\tau_{i_k}|(T, E, O) \quad \text{(by Definition 4.8)} \\
  \text{iff } w \models \tau_{i_1}(n) \text{ or } \cdots \text{ or } w \models \tau_{i_k}(n) \quad \text{(by the first item)} \\
  \text{iff } w \models (\tau_{i_1}(n) \lor \cdots \lor \tau_{i_k}(n)) \quad \text{(by the ES semantics)} \\
  \text{iff } w \models \tau(n) \quad \text{(since } w \models (4.19))
\]

- \( \tau = \text{Object} \) and \( \text{Object} \) is not primitive:

  \[
n \in |\text{Object}|(T, E, O) \\
  \text{iff } (n: \tau') \in O \text{ for some primitive } \tau' \in T \quad \text{(by Definition 4.8)} \\
  \text{iff } w \models \tau'(n) \text{ for some primitive } \tau' \in T \quad \text{(as in the first item)} \\
  \text{iff } w \models \text{Object}(n) \quad \text{(since } w \models (4.20))
\]

Furthermore, \( \text{Map}_0 \) ensures that the truth of static formulas corresponds to that of the original ADL state:

Lemma 4.20. Let \( S \) be an ADL state with respect to \( \langle T, E, O, F \rangle \), \( \phi \) be a precondition formula with respect to \( \langle T, E, O, F, \emptyset \rangle \), and \( w \) any world such that \( w \models \text{Map}_0(T, E, O, F, S) \). Then

\[
S \models \phi \iff w \models \phi.
\]

Proof. The proof is by induction over the structure of \( \phi \):
• $\phi = F_1(\vec{c})$:

\[
S \models F_1(\vec{c})  \\
\text{iff } F_1(\vec{c}) \in S \quad \text{(by the ADL semantics)}  \\
\text{iff } \vec{c} \text{ is one of the } \vec{o}_j \text{ in (4.17)} \quad \text{(by assumption)}  \\
\text{iff } w \models F_1(\vec{c}) \quad \text{(since } w \models (4.17))
\]

• $\phi = (c_1 = c_2)$:

\[
S \models (c_1 = c_2)  \\
\text{iff } c_1 \text{ and } c_2 \text{ are identical} \quad \text{(by the ADL semantics)}  \\
\text{iff } w \models (c_1 = c_2) \quad \text{(by the ES semantics)}
\]

• $\phi = \phi_1 \land \phi_2$:

\[
S \models \phi_1 \land \phi_2  \\
\text{iff } S \models \phi_1 \text{ and } S \models \phi_2 \quad \text{(by the ADL semantics)}  \\
\text{iff } w \models \phi_1 \text{ and } w \models \phi_2 \quad \text{(by induction)}  \\
\text{iff } w \models \phi_1 \land \phi_2 \quad \text{(by the ES semantics)}
\]

• $\phi = \neg \phi'$:

\[
S \models \neg \phi'  \\
\text{iff } S \not\models \phi' \quad \text{(by the ADL semantics)}  \\
\text{iff } w \not\models \phi' \quad \text{(by induction)}  \\
\text{iff } w \models \neg \phi' \quad \text{(by the ES semantics)}
\]

• $\phi = \forall x: \tau. \phi'$:

\[
S \models \forall x: \tau. \phi'  \\
\text{iff } S \models \bigwedge_{o \in \tau(T,E,0)} \phi'_{\tau,o} \quad \text{(by the ADL semantics)}  \\
\text{iff } S \models \phi'_{\tau,o} \text{ for all } o \in \tau(T,E,0) \quad \text{(by the ADL semantics)}  \\
\text{iff } w \models \phi'_{\tau,o} \text{ for all } o \in \tau(T,E,0) \quad \text{(by induction)}  \\
\text{iff } w \models \phi'_{\tau,o} \text{ for all } o \text{ such that } w \models \tau(o) \quad \text{(by Lemma 4.19)}  \\
\text{iff } w \models \forall x. \tau(x) \supset \phi' \quad \text{(by the ES syntax and semantics)}
\]
4.2 The ADL Fragment

We proceed to the central lemma of this section, showing that the BAT given by Map induces the same truth values for fluent atoms in the successor situation as the corresponding update to the ADL state:

Lemma 4.21. Let

1. $D = (T, E, D_P, F, \Lambda)$ be an ADL domain;
2. $P = (D, O_P, I, G)$ be an ADL problem;
3. $\Sigma = \Sigma_0 \cup \Sigma_{def} \cup \Sigma_{post} = \text{Map}(P)$;
4. $g_r(\bar{\sigma})$ be a ground action with respect to some operator $\omega_r = (\varphi_r, \pi_r, \epsilon_r) \in \Lambda$;
5. $I'$ be the state update for $I$ with respect to $g_r(\bar{\sigma})$;
6. and $w$ be any world such that $w \models \Sigma$.

Then 

$$w \models [p]F_i(\bar{n}) \iff F_i(\bar{n}) \in I'.$$

Proof.

$w \models [p]F_i(\bar{n})$

iff $w[F_i(\bar{n}), p] = 1$ (by the $\mathcal{ES}$ semantics)

iff $w \models \gamma_{F_i,\bar{n}} a_p$ (since $w \models \Sigma_{post}$ by assumption)

iff $w \models \gamma_{F_i,\bar{n}} a_g(\bar{\sigma})$ (since $p = [g_r(\bar{\sigma})]_w$ by assumption)

iff $w \models (\gamma_{F_i,\bar{n}} + a_g(\bar{\sigma})) \land \pi_{F_i}(\bar{n}) \lor F_i(\bar{n}) \land - (\gamma_{F_i,\bar{n}} - a_g(\bar{\sigma}))$ (by the definition of $\gamma_{F_i}$ as in (4.26))

iff $w \models (\gamma_{F_i,\bar{n}} + a_g(\bar{\sigma}))$ and $w \models \pi_{F_i}(\bar{n})$ or $w \models F_i(\bar{n})$ and $w \not\models (\gamma_{F_i,\bar{n}} - a_g(\bar{\sigma}))$ (by the $\mathcal{ES}$ semantics)

iff $w \models \bigvee \exists y_j.g_r(\bar{\sigma}) = g_j(\bar{y_j}) \land (\gamma_{F_i,\bar{n}} + a_g(\bar{\sigma}))$ and $w \models \pi_{F_i}(\bar{n})$

or $w \models F_i(\bar{n})$ and $w \not\models \bigvee \exists y_j.g_r(\bar{\sigma}) = g_j(\bar{y_j}) \land (\gamma_{F_i,\bar{n}} - a_g(\bar{\sigma}))$

where the first disjunction ranges over all the positive effects ($\forall \exists x_i.\pi_{F_i}. \gamma_{F_i,g_j}^+ \Rightarrow F_i(\bar{x_i})$)

and the second one over all the negative effects ($\forall \exists x_i.\pi_{F_i}. \gamma_{F_i,g_j}^- \Rightarrow - F_i(\bar{x_i})$)

(by the definitions of $\gamma_{F_i}^+$ and $\gamma_{F_i}^-$ in (4.24) and (4.25), respectively)

iff $(\forall \exists x_i \cdot \pi_{F_i}. \gamma_{F_i,g_j}^+ \Rightarrow F_i(\bar{x_i})) \in \epsilon_r$ s.t. $w \models (\gamma_{F_i,\bar{n}} + a_g(\bar{\sigma}))$ and $w \models \pi_{F_i}(\bar{n})$

or $w \models F_i(\bar{n})$ and

for all $(\forall \exists x_i \cdot \pi_{F_i}. \gamma_{F_i,g_j}^- \Rightarrow - F_i(\bar{x_i})) \in \epsilon_r$, $w \not\models (\gamma_{F_i,\bar{n}} - a_g(\bar{\sigma}))$

(by unique names for actions)
Let further

\[ \text{Theorem 4.22.} \]

By Lemma 4.18, \( \Sigma \)

Proof. Let \( w \in I \) and \( \bar{n} \in |\tau_F|_I \).

(4.11) and (4.12)

by (4.13)

Finally, we have everything at hand for our main theorem: That indeed the updated ADL state corresponds to the progression of the basic action theory \( \text{Map}(\mathbb{P}) \):

\[ \text{Theorem 4.22.} \]

1. \( D = (T, E, 0, F, \lambda) \) be an ADL domain;

2. \( P = (D, 0_P, I, G) \) be an ADL problem;

3. \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} = \text{Map}(\mathbb{P}) \);

4. \( g_r(\lambda) \) be a ground action with respect to some operator \( \omega_r \in \lambda \);

5. and \( I' \) be the state update for \( I \) with respect to \( g_r(\lambda) \).

Let further \( \Sigma' \) denote \( \text{Map}_0(T, E, 0, \cup 0_P, F, I') \). Then \( \Sigma' \) is a progression of \( \Sigma_0 \) through \( g_r(\lambda) \) with respect to \( \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \).

Proof. By Lemma 4.18, \( \Sigma' \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) is a basic action theory, which means that \( \Sigma' \) is a set of fluent sentences, and hence has the right form for a progression. According to Definition 3.37, we are left to show that for any world \( w' \), \( w' \models \Sigma' \) iff there is a world \( w \) such that \( w \models \Sigma \) and \( w'_{\Sigma} = w_p \), where \( p = |g_r(\lambda)|_{w} \).

\[ \text{“⇒”}: \] Let \( w \models \Sigma, w'_{\Sigma} = w_p, \text{ and } p = |g_r(\lambda)|_{w}. \) We have to show that \( w' \models \Sigma_0 \).

- \( w' \models (4.17) \) (constructed using \( I' \)):

\[ w' \models F_i(\lambda) \]

iff \( w'[F_i(\lambda), \langle \rangle] = 1 \) (by the ES semantics)

iff \( w'_{\Sigma} [F_i(\lambda), \langle \rangle] = 1 \) (by Definition 3.31)
4.2 The ADL Fragment

iff $w_p[F_i(\vec{c}), \langle \rangle] = 1$ (by assumption)
iff $w[F_i(\vec{c}), p] = 1$ (by Definition 3.35)
iff $F_i(\vec{c}) \in I'$ (by Lemma 4.21)

• $w' \models (4.18)$:

  $w' \models \tau_i(c)$
  iff $w'[\tau_i(c), \langle \rangle] = 1$ (by the ES semantics)
  iff $w'_p[\tau_i(c), \langle \rangle] = 1$ (by Definition 3.31)
  iff $w_p[\tau_i(c), \langle \rangle] = 1$ (by assumption)
  iff $w[\tau_i(c), p] = 1$ (by Definition 3.35)
  iff $w[\tau_i(c), \langle \rangle] = 1$ (by the rigidity constraint)
  iff $w' \models \tau_i(c)$ (by the ES semantics)
  iff $c$ is among the $o_{ij}$ in (4.18) (since $w \models (4.18)$ by assumption)

• $w' \models (4.19)$:

  $w' \models \tau_i(c)$
  iff $w \models \tau_i(c)$ (as above)
  iff $w \models \tau_{ij}(c)$ for one of the $\tau_{ij}$ (since $w \models (4.19)$ by assumption)
  iff $w' \models \tau_{ij}(c)$ for one of the $\tau_{ij}$ (again as above, in reverse order)

• $w' \models (4.20)$:

  $w' \models Object(c)$
  iff $w \models Object(c)$ (as above)
  iff $w \models \tau_j(c)$ for some prim. $\tau_j \in T$ (since $w \models (4.20)$ by assumption)
  iff $w' \models \tau_j(c)$ for some prim. $\tau_j \in T$ (again as above, in reverse order)

$\Rightarrow$: Let $w' \models \Sigma'_0$. We construct $w$ as follows. First let $w''$ be a world such that for all the $F_i$ and all $\vec{n}$,

(4.27) $w''[F_i(\vec{n}), \langle \rangle] = 1$ iff $F_i(\vec{n}) \in I'$
and for all the \( \tau_i \) and all \( n \),

\[
(4.28) \quad w''[\tau_i(n), ()] =
\begin{cases}
1, & \text{if } \tau_i \in T \text{ is primitive and } (n:\tau_i) \in \emptyset \\
1, & \text{if } (\tau_i:(\text{either } \tau_{i_1} \cdots \tau_{i_k})) \in E \text{ and } (n:\tau_{i_j}) \in \emptyset \text{ for some } j \text{ with } 1 \leq j \leq k_i \\
1, & \text{if } \tau_i = \text{Object} \text{ and } (n:\tau_j) \in \emptyset \text{ for some } j \\
0, & \text{else}
\end{cases}
\]

For all other predicates, let

\[
(4.29) \quad w''[P(\bar{n}), p \cdot z] = w'[P(\bar{n}), z],
\]

and similarly for all primitive terms,

\[
(4.30) \quad w''[t, p \cdot z] = w'[t, z].
\]

Now let \( w \) be \( w'_{\Sigma} \). We have to show that \( w \models \Sigma \) and \( w_p = w'_\Sigma \). For the former, note that we get \( w'' \models (4.17), w'' \models (4.18), w'' \models (4.19) \) and \( w'' \models (4.20) \) (the latter only case that \( \text{Object} \) is not primitive) immediately by the construction above. Hence \( w'' \models \Sigma_0 \), and therefore by Lemma 3.32 (2), \( w \models \Sigma \). It remains to be proven that \( w_p = w'_\Sigma \), meaning that for all primitive formulas \( \beta \) and all primitive terms \( t \),

\[
(4.31) \quad w_p[\beta, z] = w'_\Sigma[\beta, z] \\
(4.32) \quad w_p[t, z] = w'_\Sigma[t, z]
\]

Since we do not use any functional fluents here, the primitive terms can be handled quickly as follows:

\[
w_p[t, z] \\
= w[t, p \cdot z] \quad (\text{by Definition 3.35}) \\
= w'[t, p \cdot z] \quad (\text{by Definition 3.31, since no functional fluents in } \mathcal{D} \cup \mathcal{F}) \\
= w'[t, z] \quad (\text{by assumption (4.30)}) \\
= w'_\Sigma[t, z] \quad (\text{again by Definition 3.31})
\]

As an immediate consequence we get that for any ground term \( t \),

\[
(4.33) \quad |t|_{w_p} = |t|_{w'_\Sigma}.
\]

Furthermore, in the exact same manner as for the primitive terms, we can prove \( w_p[P(\bar{n}), z] = w'_\Sigma[P(\bar{n}), z] \) for all primitive formulas that involve predicates \( P \not\in \mathcal{D} \cup \mathcal{F} \cup \{\tau_1, \ldots, \tau_k, \text{Object}\} \).

The types \( \tau \), including \( \text{Object} \), can be treated as follows:
It remains to be shown that $w_p$ and $w'_\Sigma$ agree on all primitive formulas involving one of the $F_i$ or $\text{Poss}$. For this matter, we prove a more general property, namely that for any fluent sentence $\phi$ with respect to $\langle D, F \rangle$,

\[(4.34) \quad w_p, z \models \phi \iff w'_\Sigma, z \models \phi.\]

The proposition to be proven then corresponds to the special case where $\phi = F_i(\vec{n})$. The proof is by an outer induction on the length of $z$, and a sub-induction on the structure of $\phi$. Note that below the distinction between the case $z = \langle \rangle$ and the case $z = z' \cdot p'$ for the outer induction is only necessary for $\phi = F_i(\vec{t})$. For all the other constructs and symbols, the two cases can be proven identically.

- $\phi = F_i(t_1, \ldots, t_{m_i}), F_i \in F$, case $z = \langle \rangle$:

  \[
  w_p, \langle \rangle \models F_i(t_1, \ldots, t_{m_i})
  \]

  iff $w_p[F_i(n_{1}, \ldots, n_{m_i}), \langle \rangle] = 1$, where $n_j = |t_j|_{w_p}$ (by Definition 3.35)

  iff $w_p[F_i(n_{1}, \ldots, n_{m_i}), \langle \rangle] = 1$, where $n_j = |t_j|_{w'_\Sigma}$ (by Definition 3.35)

  iff $F_i(n_{1}, \ldots, n_{m_i}) \in I'$, where $n_j = |t_j|_{w'_\Sigma}$ (by Lemma 4.19)

  iff $w'[F_i(n_{1}, \ldots, n_{m_i}), \langle \rangle] = 1$, where $n_j = |t_j|_{w'_\Sigma}$ (by Definition 3.35)

  iff $w'_\Sigma[F_i(n_{1}, \ldots, n_{m_i}), \langle \rangle] = 1$, where $n_j = |t_j|_{w'_\Sigma}$ (by Definition 3.35)

- $\phi = F_i(t_1, \ldots, t_{m_i}), F_i \in F$, case $z = z' \cdot p'$:

  \[
  w_p, z \models F_i(\vec{t})
  \]

  iff $w_p[F_i(\vec{n}), z' \cdot p'] = 1$, where $\vec{n} = |\vec{t}|_{w_p}^{z' \cdot p'}$ (by the ES semantics)

  iff $w_p[F_i(\vec{n}), z' \cdot p'] = 1$, where $\vec{n} = |\vec{t}|_{w'_\Sigma}^{z' \cdot p'}$ (by Definition 3.35)

  iff $w[F_i(\vec{n}), p \cdot z' \cdot p'] = 1$, where $\vec{n} = |\vec{t}|_{w'_\Sigma}^{z' \cdot p'}$ (by Definition 3.35)
iff \( w, p \cdot z' \models (\gamma F_i)_{\bar{t}'}^{\bar{z}'}_p \), where \( \bar{n} = |\bar{t}'\bar{z}'_w\Sigma \) (since \( w \models \Sigma_{\text{post}} \))

iff \( w, p' \models (\gamma F_i)_{\bar{t}'}^{\bar{z}'}_p \), where \( \bar{n} = |\bar{t}'\bar{z}'_w\Sigma \) (by Lemma 3.36 (2))

iff \( w', z' \models (\gamma F_i)_{\bar{t}'}^{\bar{z}'}_p \), where \( \bar{n} = |\bar{t}'\bar{z}'_w\Sigma \) (by the outer induction)

iff \( w'_\Sigma[F_i(\bar{n}), z' \cdot p'] = 1 \), where \( \bar{n} = |\bar{t}'\bar{z}'_w\Sigma \) (since \( w'_\Sigma \models \Sigma_{\text{post}} \))

iff \( w'_\Sigma \models F_i(\bar{t}) \) (by the ES semantics)

\[ \text{iff } w, p \models G(t_1, \ldots, t_k), \text{ } G \text{ rigid:} \]

\[ \begin{align*}
    w, p, z & \models G(t_1, \ldots, t_k) \\
    \text{iff } & \ w_p[G(n_1, \ldots, n_k), z] = 1, \text{ where } n_j = |t_j|_{w_p}^{\bar{z}} \\
    \text{iff } & \ w_p[G(n_1, \ldots, n_k), z] = 1, \text{ where } n_j = |t_j|_{w'_\Sigma}^{\bar{z}} \\
    \text{iff } & \ w'_\Sigma[G(n_1, \ldots, n_k), z] = 1, \text{ where } n_j = |t_j|_{w'_\Sigma}^{\bar{z}} \\
    \text{iff } & \ w'_\Sigma, z \models G(t_1, \ldots, t_k) \\
\end{align*} \]

(by the ES semantics)

(by (4.33))

(by induction)

(by the ES semantics)

(by the ES semantics)

(by the ES semantics)

(by the ES semantics)

In the third step we made use of the fact that the proposition (4.31) has already been proven further above for all primitive formulas \( \beta \) involving types \( \tau \) or some predicate \( P \notin D \cup F \cup \{ \tau_1, \ldots, \tau_k, \text{Object} \} \).

\[ \text{\bullet } \phi = (t_1 = t_2): \]

\[ \begin{align*}
    & w, p \models (t_1 = t_2) \\
    \text{iff } & \ |t_1|_{w_p}^{\bar{z}} = |t_2|_{w_p}^{\bar{z}} \\
    \text{iff } & \ |t_1|_{w'_\Sigma}^{\bar{z}} = |t_2|_{w'_\Sigma}^{\bar{z}} \\
    \text{iff } & \ w'_\Sigma, z \models (t_1 = t_2) \\
\end{align*} \]

(by the ES semantics)

(by (4.33))

(by the ES semantics)

\[ \text{\bullet } \phi = \phi_1 \land \phi_2: \]

\[ \begin{align*}
    & w, p \models \phi_1 \land \phi_2 \\
    \text{iff } & \ w, p, z \models \phi_1 \text{ and } w, p, z \models \phi_2 \\
    \text{iff } & \ w'_\Sigma, z \models \phi_1 \text{ and } w'_\Sigma, z \models \phi_2 \\
    \text{iff } & \ w'_\Sigma, z \models \phi_1 \land \phi_2 \\
\end{align*} \]

(by the ES semantics)

(by induction)

(by the ES semantics)

\[ \text{\bullet } \phi = \lnot \phi': \]

\[ \begin{align*}
    & w, p \models \lnot \phi' \\
    \text{iff } & \ w, p, z \not\models \phi' \\
\end{align*} \]

(by the ES semantics)
iff \( w'_{\Sigma}, z \not\models \phi' \)  
iff \( w'_{\Sigma} \models \neg \phi' \) (by induction)

\( \phi = \forall x \phi' \):

\( w_p, z \models \forall x \phi' \)

iff \( w_p, z \models \phi'^x_n \) for all \( n \in \mathbb{N}_x \) (by the \( \mathcal{E} \mathcal{S} \) semantics)

iff \( w'_p, z \models \phi'^x_n \) for all \( n \in \mathbb{N}_x \) (by induction)

iff \( w'_{\Sigma}, z \models \phi'^x \) (by the \( \mathcal{E} \mathcal{S} \) semantics)

Finally, we show that (4.31) also holds for primitive formulas with \( \text{Poss} \):

\( w_p[\text{Poss}(n), z] = 1 \)

iff \( w[\text{Poss}(n), p \cdot z] = 1 \) (by Definition 3.35)

iff \( w, p \cdot z \models \varphi^a_{\text{Poss}_n} \) (since \( w \models \Sigma_{\text{def}} \) by assumption)

iff \( w_p, z \models \varphi^a_{\text{Poss}_n} \) (by Lemma 3.36 (2))

iff \( w'_p, z \models \varphi^a_{\text{Poss}_n} \) (by (4.34))

iff \( w'_p[\text{Poss}(n), z] = 1 \) (since \( w'_p \models \Sigma_{\text{def}} \) by Lemma 3.32 (2))  \( \square \)

We note that the above result also implies that executability and validity of plans is preserved by our mapping:

**Corollary 4.23.** Let \( \mathcal{P} = \langle g_{j_1}(\bar{o}_1), \ldots, g_{j_k}(\bar{o}_k) \rangle \) be an ADL plan for the problem \( \mathcal{P} \). Then

\( \mathcal{P} \) is executable iff Map\( (\mathcal{P}) \models [g_{j_1}(\bar{o}_1)] \cdots [g_{j_l}(\bar{o}_l)]\text{Poss}(g_{j_{l+1}}(\bar{o}_{l+1})) \) for all \( l \) with \( 0 \leq l < k \)

\( \mathcal{P} \) is valid iff Map\( (\mathcal{P}) \models [g_{j_1}(\bar{o}_1)] \cdots [g_{j_k}(\bar{o}_k)]G \)

**Proof.** Obvious from Definition 4.13, Lemma 4.20, and Theorems 3.38 and 4.22.  \( \square \)

**Example 4.24.** The following sentences comprise a progression of the initial theory given in Example 4.15 through goto(hallway, officeann) with respect to the precondition axiom presented in Example 4.16 and the successor state axioms shown in Example 4.17:

\[ At(x_1, x_2) \equiv (x_1 = \text{robot} \land x_2 = \text{officeann} \lor x_1 = \text{cup}_1 \land x_2 = \text{officeann} \lor \]
\[ x_1 = \text{cup}_2 \land x_2 = \text{kitchen}) \]

\[ Holding(x_1) \equiv (x_1 = \text{cup}_1) \]

\[ Portable(x_1) \equiv (x_1 = \text{cup}_1 \lor x_1 = \text{cup}_2) \]
Nextto($x_1, x_2$) \equiv (x_1 = \text{officeann} \land x_2 = \text{hallway} \lor x_1 = \text{officebob} \land x_2 = \text{hallway} \lor x_1 = \text{kitchen} \land x_2 = \text{hallway})

Robot(x) \equiv (x = \text{robot})

Location(x) \equiv (x = \text{officeann} \lor x = \text{officebob} \lor x = \text{kitchen} \lor x = \text{hallway})

Cup(x) \equiv (x = \text{cup}_1 \lor x = \text{cup}_2)

Letter(x) \equiv \bot

Physob(x) \equiv (\text{Robot}(x) \lor \text{Cup}(x) \lor \text{Letter}(x))

Object(x) \equiv (\text{Location}(x) \lor \text{Robot}(x) \lor \text{Cup}(x) \lor \text{Letter}(x))

4.2.5 Expressivity

As mentioned earlier, while the above presented mapping is useful in that it can be seen as a declarative semantics for PDDL expressed in ES, embedding PDDL planners into Golog actually requires to translate in the other direction. Our approach is to assume that the involved axioms in the basic action theory $\Sigma$ are of a form that could result as the output of the $\text{Map}$ operation, and then simply take the inverse of $\text{Map}$ to translate a planning subproblem in Golog to a corresponding PDDL problem instance admitting the same plans. However, it is not guaranteed that we thus cover the largest possible class of translatable action theories. For this purpose, it is also necessary to consider possible translations in the backward direction, and develop a deeper understanding of how the two formalisms relate in terms of expressivity.

Nebel’s compilation scheme framework [Neb00], which he uses to compare different subsets of propositional ADL, provides a means to analyze the expressivity of planning formalisms in a formal manner. The underlying idea is to measure the relative expressiveness of two formalisms considering possible compilations of one into the other. As opposed to polynomial many-one reductions as they are often used in complexity theory, it is not required that the compilation is carried out in polynomial time, but can indeed use arbitrary computational resources. However, the result should be expressible in polynomial space. Moreover, another requirement is that the compilation works only on the domain description (i.e. operators and symbols), but is independent of the individual instance (i.e. initial state and goal). Of course, compilations have to be solution preserving, i.e. there is a plan for the original task iff there is a plan for the result task. The relative expressivity of two planning formalisms is then measured in terms of the length of the generated plans. If it is possible to come up with a compilation scheme where the size of the shortest plan for the result task is bounded linearly by the size of an optimal plan for the source task, it is concluded that the target formalism is at least as expressive as the source
formalism. Otherwise, if plans necessarily have to grow faster than linear, the source formalism is more expressive.

Based on earlier work [ENLC06], Röger et al. [RN07, RHN08] analyzed the relative expressiveness between basic action theories of the classical Situation Calculus and the ADL fragment of PDDL. It turned out that compilation schemes always directly provide a polynomial-time translation to PDDL and that the resulting plans can directly be interpreted as solution situations for the source task. They then identified a maximal subset of BATs that has equal expressiveness as the ADL fragment, and that is given by those action theories that provide full information on the initial state in an explicit (non-compact) form. More precisely, the corresponding class of BATs can roughly be characterized as those whose initial theory consists of exactly the following:

1. For each relational fluent and rigid predicate $F$, an expression of the form

$$F(x_1, \ldots, x_n, S_0) \equiv (x_1 = c_{11} \land \cdots \land x_n = c_{1n} \lor \cdots \lor x_1 = c_{m1} \land \cdots c_{mn});$$

2. a similar expression for each function except constants and action functions with object arguments;

3. unique names axioms $c_i \neq c_j$ for each pair $c_i, c_j$ of distinct constants;

4. optionally, a domain closure axiom of the form

$$\forall x. \; x = c_1 \lor \cdots \lor x = c_k.$$ 

That is to say that no incomplete knowledge whatsoever is allowed in the initial theory of the BAT, but the initial instances of fluents and rigids have to be enumerated explicitly. Consequently, even the presence of a single unary fluent with an unknown initial value lifts the corresponding BAT beyond what the expressiveness of PDDL’s ADL fragment. For a similar reason, functions may not be used in a way that they introduce new (unnamed) objects. Furthermore, since in the classical Situation Calculus, distinct constants may refer to the same individual, which is not possible in PDDL, unique name axioms are required. Interestingly, the domain closure axiom may be omitted, although PDDL makes the domain closure assumption, whereas the Situation Calculus in general does not. The reason is that the other requirements on the initial database are so restrictive that unmentioned objects in the action theory have only a minor influence on possible plans. Indeed, similar to the Universal Generalization principle presented in Section 3.2.3 or the Representation Theorem discussed in Section 3.5.3, all unmentioned objects can be treated interchangeably, and thus only models with up to $k$ additional objects have to be considered, where $k$ is the maximal number of nested quantifiers.
Considering the form of initial theory axioms (4.17) to (4.20) as given in Section 4.2.3, it becomes apparent that our Map operation yields a very close, though not exact approximation of the set of translatable BATs. Obviously, support for rigid predicates and functional fluents can be added in a straightforward manner by simulating them through relational fluents. Moreover, as long as we resort to standard names instead of constant symbols, unique names axioms are of course unnecessary. The most convenient fact for the embedding of PDDL planners into Golog is that we do not have to worry about the domain closure assumption. Note that the above stated domain closure axiom is unsatisfiable in $\mathcal{ES}$, and that we instead ensured finite domains by resorting to typed quantification in preconditions and conditional effects. It follows from the above that we do not have to enforce this special form of quantification when translating to PDDL as long as it is ensured that the agent possesses complete knowledge about all fluents and rigids involved in a planning subtask.

4.3 Evaluation

The previously presented theoretical results provide insights on the semantical compatibility between Golog and PDDL and how they compare in terms of expressiveness. In this section, we complement these results with an empirical evaluation that shows that equipping a Golog system with a PDDL planner pays off in terms of the runtime performance of the overall system. For that matter, we study a number of example application domains and compare the needed computation times for varying problem sizes and difficulties.

4.3.1 Embedding PDDL Planning into Golog

IndiGolog

Out of the many Golog variants that were discussed in Section 2.1 and among those formally defined in Section 3.6, we chose to use an implementation of IndiGolog [GLLS04]. The latter possesses a number of features that makes it particularly suited for many practical scenarios. First, it works in cases where agents only have incomplete knowledge about the state of the world, and where sensing actions can and most often have to be used in order to gather further information that is required to fulfill the task. Furthermore, so-called exogenous actions reflect changes in a dynamic environment that are not caused by an action of the agent.

Moreover, an important aspect about the language is that programs are executed in an incremental, online manner. Unlike in the original Golog, where the system searches for an action sequence that constitutes a legal execution of the entire input program, IndiGolog does
not apply a general lookahead, but leaves it to the programmer to explicitly state which parts of the program ought to be solved by means of search so that the system puts its computational effort where it is actually needed.

For our experimental results, we resorted to a Prolog implementation of an IndiGOLOG agent system developed and maintained by Sebastian Sardiña and Stavros Vassos, and that is based on code originally written by Hector Levesque and Maurice Pagnucco. The framework is available on Sourceforge\(^2\) under an Open Source license.

The FF PDDL Planner

The FF\(^3\) planning system developed by Hoffmann [HN01] is a fully automated system for classical planning that supports the full ADL fragment of PDDL. It has proven its quality by winning the automated track of the planning competition in 2000 and being used as the basis for many state-of-the-art planners that have been developed later on.

FF performs a forward search in the state space, guided by a heuristic function that is automatically extracted from the domain description. The heuristic gets derived from the corresponding relaxed planning task that results from the original one by ignoring the delete effects of the actions. This relaxed task can be solved in polynomial time and the number of actions in the resulting plan provides an estimate for the goal distance in the original task. Furthermore, the relaxed task is used to identify so-called helpful actions that are expected to be a good choice for the next action and which hence are considered first during search. The search method used by the FF system is a variant of hill-climbing called enforced hill climbing. The main difference to the standard algorithm is that in the case of a plateau it switches to best first search until a node with a strictly better evaluation is found.

The Integration

For the integration, we implemented the previously mentioned idea that whenever a planning subproblem arises during the execution of a GOLOG program, it is translated into PDDL and the planner is called. The resulting plan is translated back and GOLOG resumes executing that plan.

In order to formalize planning subproblems, we use a special form of the achieve operator that takes a goal formula and a list of operators to be used as arguments. In terms of the

\(^2\)http://sourceforge.net/projects/indigolog/
\(^3\)http://fai.cs.umi-saarland.de/hoffmann/ff.html
IndiGOLOG constructs presented in Section 3.6.3, it is semantically defined as follows:

\[
\text{solve}(\phi, [g_1, \ldots, g_k]) \overset{def}{=} \Sigma( \text{while } \neg \phi \text{ do } \pi a \ [(a \in [g_1, \ldots, g_k])?; a] \text{ endwhile } )
\]

Note that due to the fact that IndiGOLOG executes programs online by default, we need the search operator \(\Sigma\) here in order to tell the interpreter that it indeed has to apply lookahead to find a suitable execution sequence before it starts to execute any actions. In place of the \(\text{Appropriate}(a)\) condition we simply have the requirement that the actions of the plan are chosen from the provided list:

\[
a \in [g_1, \ldots, g_k] \overset{def}{=} \bigvee_j \exists \vec{y}_j : \vec{\tau}_{g_j} \cdot a = g_j(\vec{y}_j).
\]

To compare the performance of the combined system against pure GOLOG, we considered two different implementations of \text{solve}. In one, the task is translated into a PDDL planning problem, which is then referred to FF. The other uses an internal lookahead mechanism of IndiGOLOG that solves planning problems by means of a (blind) iterative deepening search.

Both planners were of course provided with the same amount of information: a list of available actions, the fluent predicates involved (including their initial values) as well as the types of objects, fluent parameters and action parameters. Whereas the internal method directly uses the appropriate part of the GOLOG domain axiomatization, FF is given the corresponding PDDL translation as described earlier.

Since PDDL makes the closed-world assumption, we have the requirement that whenever a planning subtask is to be executed, the agent possesses complete information about all necessary fluents. For two reasons, this restriction is not as harsh as it may seem at first glance. On the one hand, it only applies to the fluents that are part of the planning subproblem, i.e. those that are mentioned in preconditions and effects of actions listed in the \text{solve} statement. This means that we really only make a local closed-world assumption [EGW97] within the context of the planning task, and that the agent may still be ignorant about other aspects of its environment. On the other hand, the values of the involved fluents may even be unknown to the agent at times, as long as it acquires the necessary information “just in time” before calling the planner. Thus, for the epistemic feasibility of our programs, it is sufficient to make a dynamic closed-world assumption [GLS01].

4.3.2 Experiments and Results

For the evaluation, we designed three example application domains. The first one is a logistics scenario in which trucks have to be used in order to transport packages between locations. In
the second domain, GOLOG is used to implement an elevator control. The third domain is a variant of our running office robot example where it is the task of the robot to transport letters between the workers' offices.

While the first two examples are inspired by similar domains used in the International Planning Competition, all three application scenarios go beyond what can be done with classical, sequential planning. This is either due to dynamic changes happening during the execution of plans, or because of the agent having to gather information at runtime by means of sensing. The details will be discussed below.

For our experiments, we have extended the INDiGOLOG framework by a simple simulator that plays the role of the outside world. It runs in a separate instance of PROLOG and communicates with GOLOG via TCP/IP sockets. The basic idea is to keep track of the (relevant part of the) world state using PROLOG's assert and retract mechanism. When an exogenous action occurs and what sensing result is returned after the execution of an action can then be defined as conditions with respect to the simulated state. All experiments were performed on a PC with an Intel Core 2 Duo E6750 CPU running at 2.66 GHz with 2 GB of memory.

**Logistics Domain**

The first domain that we studied serves as a representative for all kinds of logistics applications. The task is to transport packages to their destination locations, using a number of trucks which can only hold one package at a time. The direct connections between locations form a (not necessarily complete) graph structure. The domain has the dynamic aspect that new packages keep arriving at runtime, represented by exogenous actions, and have to be picked up and delivered in turn.

In the logistics domain, we defined a control program using prioritized interrupts:

```log
proc mainControl
  ⟨ UndeliveredPackages → deliverPackages ⟩ ⟨ ¬ Finished → wait ⟩
endProc
```

The program is to be understood as follows. In each cycle of the (implicit) main loop, if there are packages that have not yet been delivered compute a plan to deliver them and execute it. If this is not the case but execution is not yet finished, do nothing for one cycle. Otherwise terminate.

Here, Finished is a fluent that serves as a flag for signalling when program execution is supposed to halt. This is necessary in order to be able to perform finite experiments for a task that
is indeed open-ended: While delivering the currently pending packages, new delivery requests keep arriving, each of which being modelled by an exogenous action \texttt{new\_package(p,l,d)} that sets the current location of package \texttt{p} to \texttt{l} and its destination to \texttt{d}. Since the system does not know in advance when and how many new package arrivals will occur, a special exogenous action \texttt{no\_more\_packages} is used to set \texttt{Finished} to \texttt{⊤} after the last \texttt{new\_package} event indicating that the experiment ends at this point. The condition \texttt{UndeliveredPackages} is used to test whether there are still packages to be delivered:

\[
\text{UndeliveredPackages} \overset{def}{=} \exists p: \text{Package} \exists l: \text{Location} \exists d: \text{Location}. \text{At}(p,l) \land \text{Destination}(p,d) \land (l \neq d)
\]

The \texttt{deliverPackages} procedure is the part where planning comes into play:

\begin{verbatim}
proc deliverPackages
    solve ( \forall p: \text{Package} \forall d: \text{Location}. \text{Destination}(p,d) \supset \text{At}(p,d),
             [load,unload,drive] )
endProc
\end{verbatim}

For both of the planners, we considered two variants. In the first one, once a plan is found it gets executed entirely. Packages arriving during that time are ignored until plan execution finishes and the next call to the planner is made. In the other variant, the system aborts the current plan and performs a re-planning after each \texttt{new\_package} event. The latter is obtained by nesting planning instances \( \delta = \text{solve}(\phi, [g_1, \ldots, g_k]) \) within the \( \Gamma(\alpha, \delta) \) construct for guarded execution introduced in Section 3.6.3, where \( \alpha \) expresses the condition that a new package has just arrived.

We performed a series of experiments where the number of locations varied between 3 to 7, the number of trucks between 2 and 3 and the number of dynamically arriving packages among 3 and 5. For each combination, we created 10 different domain instances. The initial locations of trucks and packages as well as the destinations of the packages are chosen randomly. Two locations are connected with a probability of 50%; additional random edges ensure that the roadmap graph forms a single connected component. In each instance, there is one initial package, and the intervals between the arrival times of new packages vary between 2 and 8 steps, where one step corresponds to the execution of a primitive, non-exogenous action.

For each planner variant and domain instance we measured the overall runtime of the system and the number of steps (minus the number of \texttt{wait} actions) that were taken until termination. Runs that did not terminate within 300 seconds were aborted. The runtime includes a wait interval of 0.5 seconds after each executed action which was reserved to handle the communication with and the state update of the simulator.
Table 4.1 on page 158 summarizes the results of the experiments. The columns titled “solved” shows how many of the 10 instances were solved within the time limit by each method. Here, “P” refers to the variant where FF was used for planning while “G” means that the internal GoLog planning mechanism was used. Further “R” means the variant where instant re-planning was done after the arrival of a new request and “N” the one where the current plan was not immediately aborted. The median runtimes for each combination are given in the columns titled “times” and are moreover shown graphically in Figure 4.5, using a logarithmic scale. In those cases where the run did not finish within the timeout the value is set to the maximal time of 300 seconds. The columns titled “steps” contains the median number of steps that were taken, considering only instances that were solved by all methods.

The results clearly show that using FF instead of the internal planner has a large impact on the necessary computation time of the system, letting it solve instances within seconds which otherwise would require several minutes. Although FF is a satisficing planner, which means that it may return suboptimal plans in terms of plan length, the number of steps required by the combined system is only slightly higher than when the internal planner is used, where the latter is guaranteed to find optimal plans due to its iterative deepening strategy.
**Elevator Domain**

The second test domain has been inspired by the miconic elevator domains of the International Planning Competition in 2000. There is an elevator moving between the floors of a building. At some floors passengers are waiting and should be transported to their respective destination floor. During the program execution new passengers arrive randomly.

There are three sorts of actions that can be used to serve the passengers:

- Movement actions move the elevator from one floor to an adjacent floor.
- Since an elevator can move faster if it does not have to stop at each floor, there are also actions for fast movements that overcome two floors within one step.
- The third sort of actions are the stop actions which cause all passengers waiting at the current floor to enter the elevator and drop off all boarded passengers whose destination is the current floor.

The experimental setting for the elevator domain is analogous to the one of the logistics domain and uses the following main program:

\[
\text{proc mainControl}
\langle \text{UnservedPassengers} \rightarrow \text{servePassengers} \rangle \langle \neg \text{Finished} \rightarrow \text{wait} \rangle
\]

endProc

Planning is used to serve the passengers that have not reached their destination yet:

\[
\text{proc servePassengers}
\text{solve}(\forall p: \text{Passenger}. \text{Served}(p), [\text{move\_fast}, \text{move}, \text{stop}])
\]

endProc

Again, we tested two versions of `solve`, one calling the internal planner, the other calling the FF system, each with and without re-planning on the arrival of a new passenger.

For the benchmark instances of this domain we let the number of new passengers vary among 3, 5, 7 and 9 and the number of floors between 5 and 8. As above, we created 10 different instances for each combination, choosing the passengers’ origins and destinations randomly. Initially there is always one passenger request and the intervals between newly arriving passengers lie between 2 and 8 steps. The timeout was once more set to 300 seconds and the interval between action executions to 0.5 seconds.

Similar to the previous domain, Table 4.2 on page 159 summarizes the results for elevator domain. The median runtimes for each combination are moreover shown graphically in Figure 4.6. Again, the variants with the external planner performed much better than the ones using the internal search method. The
re-planning strategy sometimes causes the overall number of steps to increase. This happens when a current plan is discarded to serve a new request, and when later another new request is made it turns out that following the original plan would have been less costly.

Mail Delivery Robot Domain

The third domain is a variant of a common application example [TLL+97] of a mobile robot operating in an office environment, where it has to deliver letters and parcels between the workers’ mailboxes. Here, the structure of the building is assumed to consist of a number of hallways, which are connected (e.g. by an elevator) to other hallways, and where there is a certain number of offices at each hallway. Each office may contain one or multiple different mailboxes, each of which serving for both incoming and outgoing mails.

This domain involves sensing since the robot must look into a mailbox in order to find out how many and which letters it currently contains. Furthermore, before the agent actually knows
where to deliver a letter, it has to pick it up and read off the addressee. The mail delivery robot
is controlled as follows:

\[\begin{align*}
\text{proc } \text{mainControl} & \\
& \text{while } \neg \text{Finished do} \\
& \quad (\pi_r, m: \text{Mailbox}) \text{ getLettersFrom}(m); \text{ deliverLetters} \\
& \text{endWhile} \\
\text{endProc}
\end{align*}\]

Here, \(\pi_r\) denotes a variant of the non-deterministic choice of argument where the argument’s
instantiation is picked randomly. In case of the normal \(\pi\) construct, IndiGOLOG otherwise
instantiates the variable always with the first applicable symbol, which would cause the program
above to pick the same mailbox in each cycle of the loop. Once the next mailbox that should
be visited is chosen, the path to it is determined by means of planning:

\[\begin{align*}
\text{proc } \text{getLettersFrom}(m) & \\
& \quad (\pi l: \text{Location}) \text{ At}(m,l)?; \text{ solve}(\text{RobotAt}(l), [\text{move}]); \text{ takeAllLetters}(m) \\
\text{endProc}
\end{align*}\]

Taking letters out of a mailbox requires sensing:

\[\begin{align*}
\text{proc } \text{takeAllLetters}(m) & \\
& \quad \text{look into}(m); \\
& \quad \text{while } \exists l: \text{Letter. In}(l,m) \text{ do} \\
& \quad \quad (\pi l: \text{Letter}) \text{ In}(l,m)?; \text{ take out}(l,m); \text{ look at}(l); \text{ look into}(m) \\
& \quad \text{endWhile} \\
\text{endProc}
\end{align*}\]

\text{look into}(m)\) is a sensing action whose outcome is a constant \(l\) denoting one of the letters in
the box (i.e. the robot can always only “see” the topmost one). Thus, the agent gets to know
that \(\text{In}(l,m)\) is currently true. In case the mailbox is empty, the return value is instead simply
the special constant “empty”. After picking up \(l\), action \text{look at}(l)\) is applicable and causes
\(\text{Addressee}(l,m')\) to become known to the agent for some mailbox \(m'\), which is the destination
of letter \(l\). For delivering the letters obtained like this, another call to the planner is made:

\[\begin{align*}
\text{proc } \text{deliverLetters} & \\
& \quad \text{solve}(\forall l: \text{Letter. } (\exists m: \text{Mailbox. Addressee}(l,m)) \supset \text{Delivered}(l), [\text{put in, move}]) \\
\text{endProc}
\end{align*}\]
Again, we studied the system’s behavior for the case in which `solve` uses the internal planner and for the case where FF is called instead. Since there are no exogenous actions in this scenario, we do not consider dynamic re-planning.

In our benchmark scenarios the number of offices varies among 4, 8 and 16, the number of hallways among 2, 4 and 8, and the number of letters among 2, 4, 8 and 16. As in the other domains we created 10 instances for each combination, the offices being connected randomly to some hallway and hallways being connected to one another in a tree-like fashion. There are as many mailboxes as offices, but they are placed randomly. Therefore, it is possible that an office contains multiple mailboxes, only one, or even none at all. The origins and addressees of the letters are also chosen randomly. Once more we used a timeout of 300 seconds and an action execution interval of 0.5 seconds.

Table 4.3 on page 160 once again summarizes the results for this domain. The median runtimes for each combination are shown graphically in Figure 4.7. The results for this domain are again quite conclusive. The controller using FF was able to solve more tasks and throughout required less computation time. In terms of steps, the two methods are comparable, but the results are somewhat erratic, which is mostly because of the randomized strategy that was used.
4.4 Discussion

4.4.1 Summary

This chapter was about planning in GOLOG. We argued that while GOLOG is an appropriate means for the overall control of a knowledge-based agent, and although GOLOG supports planning in principle, current implementations typically perform very bad when it comes to pure planning subtasks. On the other hand, there is a multitude of state-of-the-art planning systems that use various techniques and heuristics that make them highly efficient. We proceeded with an overview of the Planning Domain Definition Language PDDL that is used as input language at the International Planning Competition. Next, we addressed an important subset of PDDL, namely its ADL fragment, and showed how ADL problems can be mapped to ES basic action theories. The formal correspondence was shown in terms of the progression of the translated action theories, and the translation can thus be seen as a declarative semantics for PDDL. Moreover, it serves as the theoretical foundation for the embedding of PDDL planners into GOLOG, which we evaluated using the example of IndiGOLOG and the FF planner. For that purpose, three example application domains were considered in which classical planning subproblems arise in the course of the execution of a high-level program. A series of experiments showed that the integration of the external planner drastically increases performance in terms of computation time.

4.4.2 Comparison to Related Work

ADL was originally introduced by Pednault [Ped89, Ped94] as a “middle-ground” between the highly restricted original STRIPS and the full first-order expressiveness of the Situation Calculus. He defined a state-transitional semantics for the language in terms of additions and deletions of tuples in relations and functions of first-order Tarskian structures. He also provided a method for deriving a Situation Calculus axiomatization, but did not show the semantics correspondence in terms of progression, as was done by Lin and Reiter [LR97] in the case of STRIPS. Moreover, he already recognized that in general, a progression may not exist, but did not explore classes of formulas where this could be guaranteed. In the case of the ADL fragment of PDDL, the reason that progression is always possible is of course the fact that PDDL makes the closed-world assumption, and that all true instances of fluent predicates can be enumerated.

The idea of defining a declarative semantics for PDDL by means of ES basic action theories has been extended to larger subsets of PDDL. In his master’s thesis, Hu [Hu06] considered a fragment that includes all the numeric and temporal features introduced in PDDL 2.1 and
2.2. Among other things, the presented mapping employs a representation proposed by Pinto [Pin94] and Reiter [Rei98] where a durative process $a$ that happens in a time interval $[t_1, t_2]$ is represented in terms of two classical, instantaneous actions $\text{start}(a, t_1)$ and $\text{end}(a, t_2)$ that start and end the process, respectively. Book-keeping auxiliary fluents and their corresponding successor state axioms are used to memorize the status of currently executed processes as well as the truth values of conditions used within inter-temporal effects. Continuous change is encoded by assigning functions of time to fluents, rather than constant values. Thus, the temporal domain is expressed as a normal basic action theory, which can then be used in the usual manner regarding regression, progression, and the execution of Golog programs. These results, which are summarized in [CHL07], do not only serve as the theoretical justification for embedding temporal PDDL planners such as Tempo [CKMW07] into Golog. By encoding the new semantics in terms of constraint programming, Hu [Hu07] also obtained a new temporal planner that could solve more planning problems than the (then) state of the art.

In her diploma thesis, Han [Han09] furthermore extends the declarative semantics to also incorporate the remaining features of PDDL 2.2 and 3.0, which include plan constraints and derived predicates. To handle plan constraints, which resemble modal operators from linear-time temporal logics [Eme90], she introduced auxiliary fluents that keep track of the truth values of the constraints throughout the intermediate situations of a plan, and provided corresponding successor state axioms. She also provided two possible encodings for derived predicates, which are fluents whose values depend on other fluents, possibly including cyclic dependencies: On the one hand, the semantics of axioms for derived predicates can be expressed by means of second-order definitional axioms in $\mathcal{ES}$. On the other hand, the closed-world assumption of PDDL also allows to determine all instances of derived predicates and compile them into the initial theory and successor state axioms, thus again acquiring a standard first-order basic action theory.

Finally, Zhu [Zhu10] presents a similar mapping between the Temporal Action Logic TAL [DGKK98] and $\mathcal{ES}$. Among other things, this includes a representation of TAL’s concept of occlusion, which refers to the fact that the exact value of a fluent is unknown within the execution interval of a durative action that affects that fluent. In Zhu’s representation, this is encoded by having auxiliary predicates as guards for the successor state axioms, thus allowing the corresponding fluents to vary arbitrarily when being occluded. Since TAL is the input language for the TALplanner [DK01b] system, the mapping again serves as the theoretical justification for embedding the corresponding planner into Golog. Zhu’s evaluation shows that this not only yields a similar increase of runtime performance as in the PDDL case, but that it is also highly beneficial to make use of TALplanner’s ability to incorporate user-provided domain-dependent control knowledge for guiding its search.
Further mappings between action logics and planning formalisms have been studied. Schiffel and Thielscher [ST05, ST06] explored the formal connections between Situation Calculus and GOLOG on the one hand and Fluent Calculus and FLUX on the other hand. Drescher [DT08] presented another declarative semantics for the ADL fragment of PDDL extended with plan constraints, however formulated in the Fluent Calculus. Thielscher [Thi11] proposed a Unifying Action Calculus that encompasses a variety of existing action logics such as the Situation Calculus, the Fluent Calculus, ADL, and the Event Calculus in order to facilitate comparisons and translations between the different formalisms.

Moreover, there has been other work on the integration of GOLOG and planning. One particularly notable approach is due to Baier, Fritz and McIlraith [BFM07] who developed a polynomial-time algorithm for compiling a GOLOG program into a planning problem of PDDL 2.1. Thus, the domain-dependent procedural control knowledge that is given by the program can be exploited by any domain-independent PDDL planner. It is also conceivable to apply their method in an integrated GOLOG-PDDL system as described in this chapter, thus being able to not only refer sequential planning tasks (i.e. solve statements) to the planner, but entire GOLOG subprograms, albeit only ones throughout which the closed-world assumption is guaranteed to hold. In subsequent work, Fritz, Baier and McIlraith [FBM08] furthermore present a similar compiler that takes a ConGOLOG program and a basic action theory and produces a new basic action theory that permits exactly the execution traces of the input program. Proving properties of programs thus becomes much simpler and can in some cases be reduced to standard regression-based reasoning.

Blom and Pearce [BP10] propose to apply relaxation-based heuristics, a popular technique used in classical planners, directly within an offline GOLOG interpreter, using a relaxed variant of regression. In a certain sense, they therefore also take advantage of results from classical planning, although in a different manner. Since PDDL serves as the common interface to all planners supported by it, our embedding lets us benefit from the newest developments in state-of-the-art planning without the need to re-implement the corresponding techniques and heuristics in the GOLOG interpreter. On the other hand, Blom and Pearce’s approach considers the interpretation of the entire program, rather than being restricted to certain subproblems only. Of course, none of these approaches for integrating planning and GOLOG is exclusive, but they can be used together in a complementary fashion.

4.4.3 Future Work

With the results presented in this chapter together with the related work discussed above, the idea of mapping PDDL to $\mathcal{E}\mathcal{S}$ for embedding state-of-the-art planners into GOLOG has
more or less been studied exhaustively. The most recent extension of PDDL (now version 3.1) introduces object functions with finite domains as a new feature. However, they are merely syntactic sugar as it is easy to see that functional, non-numeric fluents can be simulated by relational ones.

Regarding possible directions for future work, it seems to be more interesting to aim at increasing Golog’s performance in situations where the closed-world assumption is not applicable due to the agent only possessing partial world knowledge, while preferably retaining as much of first-order expressiveness as possible. One promising approach is to build upon results on so-called proper+ knowledge bases [LL02], which are an expressive class of first-order theories with disjunctive information. Liu, Lakemeyer and Levesque [LLL04, LL05b] introduce the logic $\mathcal{SL}$ in which query evaluation is classically sound, yet only sometimes complete, and which is particularly suited to represent agents that have limited computational resources. Moreover, they present a reasoning service based on $\mathcal{SL}$ that is decidable for proper+ knowledge bases, and even tractable in the case that the number of variable symbols is bounded.

The first step towards applying $\mathcal{SL}$-based reasoning in Golog is to generalize the static logic $\mathcal{SL}$ to dynamic domains. In [CL09], an extension called $\mathcal{SLA}$ is introduced which resorts to regression for solving the projection problem, and which is shown to be sound in terms of ES entailments. $\mathcal{SLA}$ forms the basis for a new Golog search operator that prefers solution plans that are the computationally cheapest to discover, rather than the shortest ones. While the latter approach has some appeal to it, it is not entirely practical: As mentioned before, solely relying on regression is problematic due to the agent’s history of executed actions growing over time and because regression tends to blow up the query. An alternative extension of $\mathcal{SL}$ which is based on progression is presented by Capes [Cap10]. It would be interesting to see how his approach can be leveraged to come up with a corresponding progression-based planning construct, in particular in combination with recent results by Liu and Lakemeyer [LL09b] who show that if actions have only local effects, the progression of proper+ knowledge bases is not only guaranteed to exist, but can be computed efficiently.
<table>
<thead>
<tr>
<th>Steps</th>
<th>Time(s)</th>
<th>Pack.</th>
<th>Trucks</th>
<th>Loc.</th>
<th>Trays</th>
<th>Load.</th>
<th>PR</th>
<th>GP</th>
<th>PN</th>
<th>GN</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23.0</td>
<td>3.3</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>22.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22.0</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21.0</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19.0</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18.0</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Passengers</td>
<td>Floors</td>
<td>Tasks</td>
<td>solved</td>
<td>time(s)</td>
<td>steps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>---------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
<td>10 10 10 10</td>
<td>GN PN GR PR</td>
<td>17.0 14.0 23.5 14.5</td>
<td>16.5 16.5 15.5 16.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>10</td>
<td>8 10 8 10</td>
<td></td>
<td>22.0 12.0 35.5 13.5</td>
<td>19.0 19.0 16.0 18.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
<td>10 9 4 10</td>
<td></td>
<td>39.5 15.0 300.0 15.0</td>
<td>19.0 19.0 16.0 19.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>10</td>
<td>6 10 3 10</td>
<td></td>
<td>87.0 17.0 300.0 18.0</td>
<td>19.0 19.0 19.0 19.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>10 9 7 9</td>
<td></td>
<td>23.0 19.0 81.5 24.0</td>
<td>25.0 25.0 29.0 30.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>10</td>
<td>8 10 1 9</td>
<td></td>
<td>90.0 20.5 300.0 27.0</td>
<td>37.0 38.0 32.0 33.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>10</td>
<td>4 9 0 9</td>
<td></td>
<td>300.0 24.0 300.0 28.0</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
<td>5 7 0 10</td>
<td></td>
<td>276.5 20.5 300.0 27.5</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>10</td>
<td>7 10 1 10</td>
<td></td>
<td>79.5 28.0 300.0 26.5</td>
<td>34.0 34.0 32.0 32.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>10</td>
<td>1 10 0 10</td>
<td></td>
<td>300.0 29.5 300.0 28.0</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>10</td>
<td>0 9 0 8</td>
<td></td>
<td>300.0 33.0 300.0 31.5</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>10</td>
<td>0 10 0 10</td>
<td></td>
<td>300.0 31.5 300.0 31.5</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>10</td>
<td>4 9 0 10</td>
<td></td>
<td>300.0 33.0 300.0 31.5</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>10</td>
<td>0 9 1 10</td>
<td></td>
<td>300.0 33.5 300.0 37.0</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>10</td>
<td>0 7 0 10</td>
<td></td>
<td>300.0 33.5 300.0 43.0</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>10</td>
<td>0 8 0 9</td>
<td></td>
<td>300.0 36.5 300.0 44.5</td>
<td>– – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Elevator Domain: Solved instances, median runtime, median number of steps
Table 4.3: Mail Domain: Solved instances, median runtime, median number of steps
Chapter 5

Golog Verification

Thus far we have been concerned with the representation and control of agents. We have argued that £S is appropriate as the underlying logic and GOLOG as a language for defining the agent’s behaviour. Furthermore it was shown in the previous chapter that embedding efficient planners can help to enhance a corresponding system’s performance, thus making a GOLOG-based agent a suitable means for the control of autonomous agents such as mobile robots.

Before deploying a GOLOG program onto the actual robot and executing it in the physical world, it is often desirable if not crucial to verify that it indeed fulfills its intended purpose and meets certain requirements. In this respect, designing and implementing agents with GOLOG is no different from industrial hardware and software development in general, where verification is a critical and major part of the overall development process [BK08]. On the one hand, this is due to the fact that it is usually more costly to fix an error the later in the process it is detected. One of many cautionary and well-known examples was a bug in the floating point division unit of Intel’s Pentium II processors whose replacement caused a loss of about 475 million US dollars, not to mention the damage this had to the company’s reputation. More importantly, in many areas hardware and software reliability is directly related to the safety of human beings, for instance in missiles, spaceships, airplanes, power plants, traffic control, and medical devices, to name but a few.

There are a number of established techniques for verification. In the case of software, important approaches are peer reviewing and testing. The former refers to the process of proof-reading the uncompiled code, preferably by software engineers that were not themselves involved in the actual programming. The latter means to actually execute the software and compare the output that it produces for certain inputs with what is to be expected according to the system’s specification. Hardware verification furthermore typically includes simulation, which is a form of testing using a software model of the system and its environment, and/or emulation, which
is also a form of testing, but using a hardware prototype.

Of course, all these techniques can (and should) in some form be applied to the verification of GOLOG-based agents and robots as well. However, there is yet another technique that is especially interesting for us in this context, namely that of formal verification. This means one expresses both the system description and the specification of its desired properties in some formal (logical) manner, and then formally proves that according to the description, the system indeed satisfies the specification. This approach lends itself particularly well to knowledge-based agents because it is unnecessary to have a description of the system that is separate from the actual implementation, which is prone to errors and inaccuracy. It is rather possible to do the verification with the very same underlying logical representation that is already used for the actual control of the agent! Although there could possibly still be flaws in the agent’s design, it can be excluded that this is due to a mismatch between description and implementation, as they are one and the same.

Regarding formal verification of GOLOG programs, the first thing to note is that if the agent performs an open-ended task, such as in our envisioned mobile robot scenario, it will (at least ideally) operate indefinitely. Its corresponding control program is hence a non-terminating one. In the case of our office robot example, a program for the task of delivering coffee to the workers on request might look like this:

\begin{verbatim}
loop
  if ¬Empty(queue) then
    (πx) selectRequest(x); pickupCoffee; bringCoffee(x)
  else wait endIf
endLoop
\end{verbatim}

Here we assume that the robot internally maintains a queue of coffee requests which will be served in order of their arrival. In each cycle of an infinite loop (denoted by the loop construct), if the queue is currently not empty, the robot chooses the next pending request \(x\) (by means of the selectRequest(\(x\)) action), gets a cup of coffee from the coffee machine (pickupCoffee) and brings it to person \(x\) (bringCoffee(\(x\))). Otherwise, it waits for a short period (wait). Requests are represented by exogenous actions requestCoffee(\(x\)), where \(x\) denotes the requesting person.

Note that here we resort to a more abstract encoding of the robot to keep the example simple. The pickupCoffee action for instance is now assumed to not only denote the actual grabbing of the coffee cup, but to also include the process of moving to the coffee machine first. Similarly, bringCoffee(\(x\)) is supposed to be a delivery action that covers also the subtask of first getting to \(x\)’s location.
The following are two examples for questions that we may want to have answered for the above program before deploying it to the robot:

1. Will every request eventually be served by the robot?

2. Is it possible that no request is ever served at all?

Surprisingly, the verification of such properties for non-terminating GOLOG programs has so far received little attention within the Situation Calculus community. A notable exception is a paper by De Giacomo, Ternovska and Reiter [GTR97], from which also the above example is adapted. They show how the semantics of non-terminating processes can be defined by means of (second-order) Situation Calculus axioms, how properties such as 1 and 2 above can be expressed using (second-order) fixpoint formulas, and that it is then possible to prove the satisfaction of these properties given the aforementioned axioms.

For verifying properties of non-terminating GOLOG programs in practice, there are however at least two problems with this. The first one is in terms of representation and comprehensibility. For instance, property 1 is expressed in [GTR97] as follows:

\[ \text{Fair}(\delta_0, S_0) \overset{\text{def}}{=} (\forall x, \delta, s) \ Trans^*(\delta_0, S_0, \delta, \text{do}(\text{requestCoffee}(x), s)) \supset \ \text{EventuallyServed}(x, \delta, \text{do}(\text{requestCoffee}(x), s)) \]

Recall that \( S_0 \) denotes the initial situation, whereas a term of the form \( \text{do}(a, s) \) means the situation resulting from doing action \( a \) in situation \( s \). \( Trans^* \) refers to the transitive closure of the \( Trans(\delta, s, \delta', s') \) predicate which defines valid transitions from programs \( \delta \) and situations \( s \) to resulting programs \( \delta' \) and resulting situations \( s' \). \( \text{EventuallyServed} \) is then defined by

\[ \text{EventuallyServed}(x, \delta_1, s_1) \overset{\text{def}}{=} \mu_{P, \delta, s} \{(∃s'')s = \text{do}(\text{selectRequest}(x), s'') \lor \ ( ((∃\delta', s') Trans(\delta, s, \delta', s')) \land \ (∀\delta', s') Trans(\delta, s, \delta', s') \supset P(\delta', s'))\}(\delta_1, s_1) \]

where the notation \( \mu_{P, \delta, s} \) denotes a least fixpoint according to the following formula:

\[ (∀\vec{x})\{\mu_{P, \delta, s}ϕ(P, \vec{y})(\vec{x}) \equiv [(∀P)[(∀\vec{y})ϕ(P, \vec{y}) \supset P(\vec{y})] \supset P(\vec{x})]\} \]

We will not explain these definitions in any more detail here since they are unnecessary for our further treatment. The above formulas shall merely illustrate that formulating properties in this manner is obviously quite involved due to the heavy reliance on second-order quantification. The underlying \( μ \)-calculus and inductive fixpoint definitions are typically difficult to grasp even for
the mathematically inclined, all the more so for a GOLOG programmer and domain designer who is most likely not an expert in logic.

The second problem with existing work on the verification of nonterminating GOLOG programs is that proving properties manually, be it in the formalism of [GTR97] or a different one, is not only tedious, but also prone to errors. A much more desirable solution would be if verification could be done automatically, using an appropriate algorithm. Nowadays the automated, formal verification of nonterminating processes is typically associated with model checking, a form of formal verification whose popularity has steadily grown since its introduction in the early 1980s [CE81, CES86, QS82]. In a nutshell, the idea is to have a model of the system, its possible states and their relation among another in the form of a finite graph structure (a so-called transition system). The specification to be verified is typically expressed in terms of formulas of temporal logic. Verification is then done by checking the satisfaction of these formulas by a systematic, graph-theoretic analysis of the model. The approach can furthermore be boosted to handle very large state spaces by resorting to compact, implicit representations such as ordered binary decision diagrams (OBDD) [Bry86], which is referred to as symbolic model checking [BCM+92, McM93]. Using appropriate finite representations, it is even possible to treat infinite-state systems [BCMS01], for example by means of pushdown automata [BEM97], petri nets [Esp94], or variants of Minsky’s counter machines [Dem06], to name but a few.

In the face of this abundance of results on model checking, it is conceivable to verify GOLOG programs by an approach similar to the embedding of PDDL planners described in Chapter 4, namely by translating program and specification properties into the model checking algorithm’s input formalism, and then call the model checker on that input. However, even in the case of compact infinite-state representations, the underlying formalisms are usually chosen very carefully to ensure decidability or even tractability of the model checking method. Consequently, these formalisms are of very restricted expressiveness, in particular regarding first-order quantification, which is either supported only in a very limited fashion, or not at all. On the other hand, the first-order expressiveness of the Situation Calculus or $\mathcal{ES}$ is considered a desirable feature that one would rather not give up, but often the reason why the corresponding language was chosen in the first place. It seems preferable to be able to do the verification in the very same expressive formalism that we use for the actual control and specification of our agent.

In this chapter, we will present a solution that addresses both these issues. On the one hand, a new logic is proposed that extends $\mathcal{ES}$ by constructs that allow to express statements about GOLOG programs and their properties. By resorting to operators known from temporal logics and fixing their meaning within the semantics, we thus acquire an expressive, yet concise and readable formalism without the need for any second-order axioms.
5.1 The Logic $\mathcal{ESG}$

On the other hand, we will then discuss several algorithms for the automated verification of properties of both terminating and nonterminating $\text{GOLOG}$ programs expressed by means of this new logic. Although they clearly draw their inspiration from the classical approaches to symbolic CTL and CTL* model checking, the proposed methods do not perform model checking in the strict sense of the term as they do not operate on a single, simplified model of the system in question. Instead, they are similar in spirit to the well-known approaches to classical reasoning tasks in the Situation Calculus such as Reiter’s Regression or Lakemeyer and Levesque’s Representation Theorem: Given a theory that describes the dynamics and the knowledge of the agent, we have a meta-level operation that iteratively transforms the original query into an equivalent expression that is in a sense “easier” to answer. This is achieved by “factoring out” the non-classical aspects of the query, thus reducing the original problem to a finite number of instances of classical FOL theorem proving. In the case of verifying $\text{GOLOG}$ programs, this amounts to iteratively applying regression-based formula manipulations until a fixpoint is reached. Fixpoints are thus merely calculated at the meta-level, rather than being constructed in the language through second-order induction axioms, as was done by De Giacomo, Ternovska and Reiter. We will further establish that the proposed methods are sound and discuss under which circumstances they are also complete.

The remainder of this chapter is organized as follows. Section 5.1 introduces the new extended logic called $\mathcal{ESG}$. In Section 5.2, the so-called characteristic graphs of $\text{GOLOG}$ programs are discussed, which constitute a paramount ingredient to our verification algorithms. The first such algorithm for nonterminating programs and a substantial subclass of query properties is presented in Section 5.3, together with a discussion of the method’s soundness and completeness. Section 5.4 then deals with an extended method that is capable of treating (almost) all properties that can be expressed in $\mathcal{ESG}$. In Section 5.5 we briefly discuss the verification of terminating programs, which can be done in a similar manner. As an implementation of the presented algorithms requires a compact representation of first-order formulas, a first-order extension of binary decision diagrams (BDDs) is proposed in Section 5.6, before we conclude with Section 5.7.

5.1 The Logic $\mathcal{ESG}$

Let us first address the issue of an appropriate representation for properties of both nonterminating and terminating $\text{GOLOG}$ programs. As far as terminating programs are concerned, we have already discussed a possibility to express a form of such properties, namely by means of the $\text{Do}$ macro operator presented in Section 3.6.1. Recall that for a simple $\text{GOLOG}$ program $\delta$,
**Do**\((\delta, \alpha)\) expands to a formula that describes all and only conditions under which some successful execution of \(\delta\) exists such that afterwards \(\alpha\) holds. Lakemeyer and Levesque introduced **Do** in order to show that it is possible to define a semantics for *GOLOG* in \(\mathcal{ES}\) that is equivalent to the one defined based on the Situation Calculus \([LRL+97]\). The latter however turned out to be problematic when it comes to extensions of the language that include some form of (interleaved) concurrency, such as by means of the \(|\ |\) operator, which is why De Giacomo, Lespérance and Levesque proposed their now well-known transition semantics for *ConGOLOG* \([GLL00]\).

In Chapter 3, we defined a similar semantics for programs including concurrency, however in a meta-logical fashion using the transition relation \(\rightarrow\) and the finality predicate \(\mathcal{F}\). What we want now is to be able to express statements about programs within our logic. For expressing postconditions of programs similar to **Do**, we propose to extend the \([\cdot]\) operator to admit arbitrary programs \(\delta\), instead of only primitive actions. Staying consistent with notational conventions in modal logics, we then define the formula

\[[\delta]\alpha\]

to mean that after *any* successful execution of \(\delta\), \(\alpha\) will hold, while \(\langle\delta\rangle\alpha\) denotes the operator’s existential dual, meaning that \(\alpha\) holds after *some* execution of \(\delta\):

\[\langle\delta\rangle\alpha \overset{df}{=} \neg[\delta]\neg\alpha.\]

Since a primitive action is only a special case of a *GOLOG* program, the operator keeps its original meaning for such actions. We furthermore get that, in a sense (that we will make precise in the course of this chapter), \(\langle\delta\rangle\alpha\) is equivalent to **Do**\((\delta, \alpha)\). However, whereas the latter is only defined for simple *GOLOG* programs, the former works on any member of the *GOLOG* family whose constructs we can define via \(\rightarrow\) and \(\mathcal{F}\).

Postconditions are of course useless in the case of non-terminating programs. What we need instead is a means of expressing facts not only for single situations, but for (possibly infinite) sequences of them. As argued above, we do not consider fixpoint definitions and second-order formulas appropriate for this, but propose to resort to operators known from temporal logics, which are much easier to read and more intuitive to grasp. For this purpose, we first introduce yet another new modal operator \([\cdot]\) such that

\[[\delta]\varphi\]

denotes that for *all* possible execution traces of \(\delta\), the formula \(\varphi\) holds. Apart from the logical constructs we have used so far, the trace formula \(\varphi\) then may contain temporal subformulas of
the form \( X\psi \) (\( \psi \) holds in the next situation) and \( (\phi \ U \ \psi) \) (\( \phi \) holds until \( \psi \) holds). Again, the above operator has an existential counterpart \( \langle\langle \cdot \rangle\rangle \) that we can define by

\[
\langle\langle \delta \rangle\rangle \phi \equiv \neg [\neg \delta] \neg \phi.
\]

Note that the temporal constructs \( X \) and \( U \) are indeed sufficient for our purposes since we can define further operators as “syntactic sugar”: \( F\phi \) (read: “eventually \( \phi \)” or “finally \( \phi \)”) denotes \((T \ U \ \phi)\), and \( G\phi \) (read: “always \( \phi \)” or “generally \( \phi \)”) stands for \( \neg F\neg \phi \). If \( \delta \) is the program (5.1) together with an encoding of the domain’s exogenous actions, then the two properties mentioned earlier can be expressed as follows:

\[
(5.2) \quad [\delta]G(\text{Occ}(\text{requestCoffee}(x))) \supset F\text{Occ}(\text{selectRequest}(x)))
\]

\[
(5.3) \quad \langle\langle \delta \rangle\rangle G\neg \exists x(\text{Occ}(\text{selectRequest}(x)))
\]

where (5.2) reads “for all possible executions of \( \delta \), it is always the case that whenever a requestCoffee(\( x \)) action occurs, then eventually there will be a corresponding selectRequest(\( x \)) action”, i.e. every request will eventually be served. (5.3) is further to be read as “there is a possible execution trace of \( \delta \) such that at all points in time, no selectRequest(\( x \)) action occurs, i.e. no request is ever served. It is justified to argue that (5.2) and (5.3), which both take not more than one line, are a much more compact representation of the properties, and probably also more readable once one is familiar with the meaning of \( G \) and \( F \). Before showing how we can automatically verify such properties, we will now define this new logic, which we call \( \mathcal{E}\mathcal{S}\mathcal{G} \) (where the \( G \) stands for GoLOG), formally.

5.1.1 Syntax

Terms are defined as before, i.e. we have variables, standard names and compound terms which come in the three sorts object, action and number.

Programs

As opposed to Chapter 3 where we used programs only meta-theoretically, they now become a part of the language. Also different from the original ConGOLOG semantics [GLL00] where programs were introduced as terms whose meaning is defined axiomatically, the program constructs are now logical (built-in) symbols with a fixed meaning. The programs we consider here are the ones admitted by the following grammar:

\[
(5.4) \quad \delta ::= \ t \ | \ \alpha? \ | \ \delta_1; \delta_2 \ | \ \delta_1|\delta_2 \ | \ \pi x.\delta \ | \ \delta_1[\delta_2 \ | \ \delta^* \]
\]
That is we allow primitive actions \( t \) (where \( t \) can be any action term), tests \( \alpha ? \) (where \( \alpha \) is a static situation formula as defined below), sequence, nondeterministic branching, nondeterministic choice of argument, concurrency, and nondeterministic iteration. Recall that thus also if statements and while loops are included:

\[
\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endIf} \defeq [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]
\]

\[
\text{while } \phi \text{ do } \delta \text{ endWhile} \defeq [\phi?; \delta]^*; \neg\phi?
\]

The infinite loop for non-terminating programs furthermore is given by:

\[
\text{loop } \delta \text{ endLoop} \defeq \text{while } \top \text{ do } \delta \text{ endWhile}
\]

We will also sometimes write \( \text{loop } \delta \text{ endLoop} \) as \( \delta^\omega \). Note that it would not represent a problem to let the new logic include the entire GOLOG language as presented in Section 3.6, but to keep things simple and because of the fact that the verification algorithms we are going to discuss only work on this subset of GOLOG (for reasons that will become apparent in Section 5.2), we restrict ourselves to the above set of constructs. Throughout this chapter, unless otherwise stated, whenever we speak of a “program”, it will be implicitly assumed that this means a GOLOG program according to (5.4).

### Situation Formulas

Also for the purpose of simplicity of the language, we will only consider objective formulas in this chapter, which now come in two different “flavours”, as given by the following definition:

**Definition 5.1** (Situation Formulas). The **situation formulas** are the least set such that

- if \( t_1, \ldots, t_k \) are terms and \( P \) is a (fluent or rigid) \( k \)-ary predicate symbol, then \( P(t_1, \ldots, t_k) \) is a situation formula;
- if \( t_1 \) and \( t_2 \) are terms, then \( t_1 = t_2 \) is a situation formula;
- if \( \alpha \) and \( \beta \) are situation formulas, \( x \) is a variable, \( P \) is a (fluent or rigid) predicate symbol, \( \delta \) is a program, and \( \phi \) is a trace formula (defined below), then \( \alpha \land \beta, \neg \alpha, \forall x. \alpha, \forall P. \alpha, \Box \alpha, [\delta] \alpha, \text{ and } \llbracket \delta \rrbracket \phi \) are situation formulas.

Situation formulas are similar to objective formulas as we used them in the previous chapters: Intuitively, they express properties which are (or are not) satisfied by single situations, albeit
this may include references to other situations by means of $[\cdot]$, $\Box$, or $[\Box\cdot]$. We again include further constructs such as $\supset$ and $\exists$ as abbreviations, as before. Moreover, let

\begin{align}
5.8 & \quad \langle \delta \rangle \alpha \overset{\text{def}}{=} \neg[\delta]\neg\alpha \\
5.9 & \quad \langle\langle \delta \rangle\rangle \varphi \overset{\text{def}}{=} \neg[\langle \delta \rangle]\neg\varphi.
\end{align}

**Trace Formulas**

Furthermore, we have a new type of formulas as follows:

**Definition 5.2** (Trace Formulas). The *trace formulas* are the least set such that

- if $\alpha$ is a situation formula, then it is also a trace formula;
- if $\phi$ and $\psi$ are trace formulas and $x$ is a variable, then $\phi \land \psi$, $\neg\phi$, $\forall x.\phi$, $X\phi$, and $\phi U \psi$ are also trace formulas.

These formulas, as the name suggests, are used to talk about *traces* of situations, i.e. finite or infinite sequences of actions. We will use them for representing the temporal properties of program execution traces. In addition to the usual abbreviations, we also have

\begin{align}
5.10 & \quad F\phi \overset{\text{def}}{=} (\top U \phi) \\
5.11 & \quad G\phi \overset{\text{def}}{=} \neg F\neg\phi.
\end{align}

**Definitions**

Because of the new modal operators, we have to revise the definitions of static and bounded formulas:

**Definition 5.3** (Static Formulas). A situation formula $\alpha$ is *static* when it contains no $[\cdot]$, no $\Box$ and no $[\Box\cdot]$ operators.

**Definition 5.4** (Bounded Formulas). A situation formula $\alpha$ is *bounded* when it contains no $\Box$ operators, no $[\Box\cdot]$ operators, and $[t]$ operators only in case the argument is an atomic action $t$.

### 5.1.2 The Semantics

Worlds in $\mathcal{ES}\mathcal{G}$ are defined exactly as they are in $\mathcal{ES}$ (Definition 3.6), and so is the denotation of terms (Definition 3.7).

\footnote{Note that we present situation formulas and trace formulas in separate definitions for the sake of readability. Since they mutually depend on each other, we would actually have to define them in a single, inductive definition.}


**Programs**

Programs are interpreted as described in Section 3.6. For convenience, their semantics is repeated below for the subset of programs we employ in this chapter. Recall that a *configuration* \( \langle z, \delta \rangle \) consists of an action sequence \( z \) and a program \( \delta \), where intuitively \( z \) is the history of actions that have already been performed, while \( \delta \) is the program that remains to be executed. Since \( \mathcal{ESG} \) is purely objective, we further do not have to include epistemic states.

**Definition 5.5** (Program Transition Semantics). The transition relation \( \xrightarrow{w} \) among configurations, given a world \( w \), is the least set satisfying

1. \( \langle z, t \rangle \xrightarrow{w} \langle z \cdot p, n i l \rangle \), if \( p = |t|^z \);
2. \( \langle z, \delta_1; \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \gamma; \delta_2 \rangle \), if \( \langle z, \delta_1 \rangle \xrightarrow{w} \langle z \cdot p, \gamma \rangle \);
3. \( \langle z, \delta_1; \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \), if \( \langle z, \delta_1 \rangle \in \mathcal{F}^w \) and \( \langle z, \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \);
4. \( \langle z, \delta_1; \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \), if \( \langle z, \delta_1 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \) or \( \langle z, \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \);
5. \( \langle z, \pi_x.\delta \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \), if \( \langle z, \delta_x^n \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \) for some \( n \in \mathbb{N}_x \);
6. \( \langle z, \delta' \rangle \xrightarrow{w} \langle z \cdot p, \gamma; \delta^* \rangle \), if \( \langle z, \delta \rangle \xrightarrow{w} \langle z \cdot p, \gamma \rangle \);
7. \( \langle z, \delta_1; \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \), if \( \langle z, \delta_1 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \);
8. \( \langle z, \delta_1; \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta_1; \delta_2 \rangle \), if \( \langle z, \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \).

The set of final configurations \( \mathcal{F}^w \) of a world \( w \) is the smallest set such that

1. \( \langle z, \alpha? \rangle \in \mathcal{F}^w \) if \( w, z \models \alpha \);
2. \( \langle z, \delta_1; \delta_2 \rangle \in \mathcal{F}^w \) if \( \langle z, \delta_1 \rangle \in \mathcal{F}^w \) and \( \langle z, \delta_2 \rangle \in \mathcal{F}^w \);
3. \( \langle z, \delta_1; \delta_2 \rangle \in \mathcal{F}^w \) if \( \langle z, \delta_1 \rangle \in \mathcal{F}^w \) or \( \langle z, \delta_2 \rangle \in \mathcal{F}^w \);
4. \( \langle z, \pi_x.\delta \rangle \in \mathcal{F}^w \) if \( \langle z, \delta_x^n \rangle \in \mathcal{F}^w \) for some \( n \in \mathbb{N}_x \);
5. \( \langle z, \delta' \rangle \in \mathcal{F}^w \);
6. \( \langle z, \delta_1|\delta_2 \rangle \in \mathcal{F}^w \) if \( \langle z, \delta_1 \rangle \in \mathcal{F}^w \) and \( \langle z, \delta_2 \rangle \in \mathcal{F}^w \).

Temporal properties that we express by situation formulas refer to *traces*, as defined below.
5.1 The Logic \( \mathcal{ESG} \)

Definition 5.6 (Traces). A trace is a possibly infinite sequence of action standard names. That is, in addition to the finite action sequences that we have considered so far, we now allow for infinite sequences as well. Formally, an infinite trace \( \pi \) is given by a mapping from the natural numbers to action standard names:

\[
\pi : \mathbb{N} \rightarrow \mathcal{N}_A
\]

We will often write such a trace as \( \pi = p_1 \cdot p_2 \cdot p_3 \cdots \) in our meta-theoretic notation. Furthermore, let \( \pi^{(i)} \) stand for the finite sequence that consists of the first \( i \) elements of \( \pi \), where \( \pi^{(0)} \) is the empty sequence \( \langle \rangle \).

As a notational convention, we use \( \tau \) to denote arbitrary traces, \( z \) for finite ones (as before) and \( \pi \) for infinite ones. Moreover, let \( Z = \mathcal{N}_A^* \) be the set of all finite traces (as before), \( \Pi = \mathcal{N}_A^\omega \) the set of all infinite traces, and \( T = Z \cup \Pi \) the set of all traces.

We can now define the traces admitted by a given program:

Definition 5.7 (Traces of Programs). Let \( \overset{w}{\rightarrow}^* \) denote the reflexive and transitive closure of \( \overset{w}{\rightarrow} \). Given a world \( w \) and a finite sequence of action standard names \( z \), the set of traces of a program \( \delta \) is

\[
\begin{align*}
|\delta|_w &= \{ z' \in Z \mid \langle z, \delta \rangle \overset{w}{\rightarrow} \langle z' \cdot \delta', \delta' \rangle \text{ and } \langle z' \cdot \delta', \delta' \rangle \in F^w \} \cup \\
\{ \pi \in \Pi \mid \langle z, \delta \rangle \overset{w}{\rightarrow} \langle z \cdot \pi^{(1)}, \delta_1 \rangle \overset{w}{\rightarrow} \langle z \cdot \pi^{(2)}, \delta_2 \rangle \overset{w}{\rightarrow} \langle z \cdot \pi^{(3)}, \delta_3 \rangle \overset{w}{\rightarrow} \cdots \}
\end{align*}
\]

where for all \( i \geq 0 \), \( \langle z \cdot \pi^{(i)}, \delta_i \rangle \notin F^w \}

In words, the finite traces admitted by some \( \delta \) given \( w \) and \( z \) are those that correspond to a finite number of transitions by means of which a final configuration is reachable. Its infinite traces are given by all infinite sequences of transitions that never visit any final configuration.\(^2\)

Situation and Trace Formulas

We are now equipped to define the truth of formulas:

Definition 5.8 (Truth of Situation and Trace Formulas). Given a world \( w \in \mathcal{W} \) and a situation formula \( \alpha \), we define \( w \models \alpha \) as \( w, \langle \rangle \models \alpha \), where for any \( z \in Z \):

1. \( w, z \models F(t_1, \ldots, t_k) \) iff \( w[F(n_1, \ldots, n_k), z] = 1 \), where \( n_i = |t_i|_w \);

2. \( w, z \models (t_1 = t_2) \) iff \( n_1 \) and \( n_2 \) are identical, where \( n_i = |t_i|_w \);

\(^2\)Similar to an infinite trace, an infinite sequence of transitions \( \langle z_0, \delta_0 \rangle \overset{w}{\rightarrow} \langle z_1, \delta_1 \rangle \overset{w}{\rightarrow} \langle z_2, \delta_2 \rangle \overset{w}{\rightarrow} \langle z_3, \delta_3 \rangle \overset{w}{\rightarrow} \cdots \) is to be understood formally as a function that maps each natural number \( k \) to a configuration \( \langle z_k, \delta_k \rangle \) such that for all \( i \geq 0 \), \( \langle z_i, \delta_i \rangle \overset{w}{\rightarrow} \langle z_{i+1}, \delta_{i+1} \rangle \).
3. \( w, z \models \alpha \land \beta \) iff \( w, z \models \alpha \) and \( w, z \models \beta \);
4. \( w, z \models \neg \alpha \) iff \( w, z \not\models \alpha \);
5. \( w, z \models \forall x. \alpha \) iff \( w, z \models \alpha^x_n \) for all \( n \in \mathbb{N}_x \);
6. \( w, z \models \forall P. \alpha \) iff \( w', z \models \alpha \) for all \( w' \sim_p w \);
7. \( w, z \models \Box \alpha \) iff \( w, z \cdot z' \models \alpha \) for all \( z' \in \mathbb{Z} \);
8. \( w, z \models [\delta] \alpha \) iff for all finite \( z' \in [\delta]_w^z \), \( w, z \cdot z' \models \alpha \);
9. \( w, z \models [\delta] \phi \) iff for all \( \tau \in [\delta]_w^z \), \( w, z, \tau \models \phi \).

The truth of trace formulas \( \phi \) is defined as follows for \( w \in \mathcal{W} \), \( z \in \mathcal{Z} \), and traces \( \tau \in \mathcal{T} \):

1. \( w, z, \tau \models \alpha \) iff \( w, z \models \alpha \), if \( \alpha \) is a situation formula;
2. \( w, z, \tau \models \phi \land \psi \) iff \( w, z, \tau \models \phi \) and \( w, z, \tau \models \psi \);
3. \( w, z, \tau \models \neg \phi \) iff \( w, z, \tau \not\models \phi \);
4. \( w, z, \tau \models \forall x. \phi \) iff \( w, z, \tau \models \phi^x_n \) for all \( n \in \mathbb{N}_x \);
5. \( w, z, \tau \models \mathbf{X} \phi \) iff \( \tau = p \cdot \tau' \) and \( w, z \cdot p, \tau' \models \phi \);
6. \( w, z, \tau \models \phi \mathbf{U} \psi \) iff there is \( z' \) such that \( \tau = z' \cdot \tau' \) and \( w, z \cdot z', \tau' \models \psi \) and for all \( z'' \neq z' \) with \( z' = z'' \cdot z''' \), \( w, z \cdot z'', z''' \cdot \tau' \models \phi \).

Most of the rules for situation formulas are the same as the ones for objective formulas stated in Definition 3.8, except for the new rules \( 8 \) and \( 9 \). According to the above, a formula \( \alpha \) is true after the execution of a program \( \delta \) if \( \alpha \) holds in all situations reachable by some finite trace of \( \delta \). Among other things this implies that any formula \( \alpha \), even if it is unsatisfiable, is always true "after" a non-terminating program:

\[
\models [\text{any}^\omega] \alpha
\]

While at first glance this may seem counter-intuitive, it indeed makes sense as the \([\delta]\) operator constitutes a form of a universal quantification over the set of all finite traces of \( \delta \), which vacuously holds whenever this set is empty. The intuition behind this becomes apparent once we rephrase the reading of \([\delta] \alpha \) as "whenever program \( \delta \) terminates, \( \alpha \) will hold".

Rule \( 9 \) on the other hand says that \([\delta] \phi \) holds if the trace formula \( \phi \) is satisfied for all traces of \( \delta \), including infinite ones. As suggested by their name, trace formulas are interpreted with
respect to a trace \( \tau \) (in addition to a world \( w \) and an action sequence \( z \)), which is why the above definition contains an additional, separate set of rules for such formulas, of which the last two are the most interesting ones: Rule 5 says that \( X\phi \) holds if \( \phi \) holds after the next action of \( \tau \), while rule 6 says that \((\phi U \psi)\) is true if \( \psi \) holds after some finite (possibly empty) prefix of \( \tau \), and \( \phi \) is satisfied at all preceding situations.

5.1.3 Properties

Before getting to the verification of programs, let us first have a look at some properties of our new logic that demonstrate how \( \mathcal{ESG} \) relates to \( \mathcal{ES} \) and Golog. First of all we have that objective \( \mathcal{ES} \) is a part of \( \mathcal{ESG} \):

**Theorem 5.9.** Let \( \alpha \) be an objective \( \mathcal{ES} \) sentence (possibly with second-order quantification), \( \models_{\mathcal{ES}} \) denote validity according to Definition 3.8, while \( \models_{\mathcal{ESG}} \) denotes validity according to Definition 5.8. Then

\[
\models_{\mathcal{ES}} \alpha \iff \models_{\mathcal{ESG}} \alpha.
\]

**Proof.** The theorem can be proven by showing \( w, z \models_{\mathcal{ES}} \alpha \iff w, z \models_{\mathcal{ESG}} \alpha \) via induction on \( \alpha \), where \( \models_{\mathcal{ES}} \) denotes satisfaction according to Definition 3.8, whereas \( \models_{\mathcal{ESG}} \) denotes satisfaction according to Definition 5.8. The underlying definition of worlds, the denotation of terms as well as the semantic rules for atoms and all constructs is identical except for \([\cdot]\) and \([\cdot]\), hence the claim is obvious in these cases. Since \( \alpha \) is assumed to be a formula of \( \mathcal{ES} \), it does not contain any \([\cdot]\) at all. Furthermore, the only appearances of \([\cdot]\) are with an argument that is an atomic action term \( t \), so we are only left to show the equivalence in this case.

For an atomic action \( t \), the only possible transition rule in Definition 5.5 is the first one, i.e. \((z, t) \xrightarrow{w} (z \cdot p, \text{nil})\), if \( p = |t|^z_w \). Furthermore \((z \cdot p, \text{nil}) \in F^w \), therefore \(|t|^z_w = \{ p \}\). Then we have:

\[
\begin{align*}
\ &\ w, z \models_{\mathcal{ESG}} [t]\alpha \\
\iff &\ \text{for all } z' \in |t|^z_w, \ w, z \cdot z' \models_{\mathcal{ESG}} \alpha \quad \text{(by Definition 3.8)} \\
\iff &\ \ w, z \cdot p \models_{\mathcal{ESG}} \alpha, \ \text{where } p = |t|^z_w \quad \text{(since } |t|^z_w = \{ p \}) \\
\iff &\ \ w, z \cdot p \models_{\mathcal{ES}} \alpha, \ \text{where } p = |t|^z_w \quad \text{(by induction)} \\
\iff &\ \ w, z \models_{\mathcal{ES}} [t]\alpha \quad \text{(by Definition 5.8)}
\end{align*}
\]

Hence we get \( \models_{\mathcal{ES}} \alpha \iff \models_{\mathcal{ESG}} \alpha. \)

Next we will show the relation of the transition semantics of Golog provided in Definition 5.5 to Lakemeyer and Levesque’s *Do* macro definition [LL05a] as given in Definition 3.56. First we have the following lemma:
Lemma 5.10.

1. $z' \in \|t\|_w^z$ iff $z' = p = \|t\|_w^z$;

2. $z' \in \|\phi\|_w^z$ iff $z' = \langle \rangle$ and $w, z \models \phi$;

3. $z' \in \|\delta_1; \delta_2\|_w^z$ iff $z' = z_1 \cdot z_2$ for some $z_1 \in \|\delta_1\|_w^z$ and some $z_2 \in \|\delta_2\|_w^{z_1};$

4. $z' \in \|\delta_1; \delta_2\|_w^z$ iff $z' \in \|\delta_1\|_w^z \cup \|\delta_2\|_w^z$;

5. $z' \in \|\pi x. \delta\|_w^z$ iff $z' \in \|\delta\|_w^n$ for some $n \in \mathcal{N}_z$;

6. $z' \in \|\delta^*\|_w^z$ iff $z' = z_0 \cdot z_1 \cdots z_k$ for some $k \geq 0$, $z_0 = \langle \rangle$ and $z_{i+1} \in \|\delta\|_w^{z_0 \cdots z_i}$.

Proof.

1. Obvious from the facts that $(z, t) \not\in \mathcal{F}_w$, that $(z, t) \xrightarrow{w} (z \cdot p, \text{nil})$ with $p = \|t\|_w^z$ is the only transition step for $t$ in Definition 5.5, and that $(z, \text{nil}) \in \mathcal{F}_w$.

2. This item is a direct consequence of the facts that there is no transition rule for $\phi$? in Definition 5.5 and that $(z, \phi) \in \mathcal{F}_w$ only in case that $w, z \models \phi$.

3. We prove this item by induction on the length of $z'$. We will use the following direct consequence of Definition 5.7:

\[
(5.14) \quad p \cdot z' \in \|\delta\|_w^z \text{ iff } \langle z, \delta \rangle \xrightarrow{w} \langle z \cdot p, \delta' \rangle \text{ and } z' \in \|\delta'\|_w^p.
\]

- $z' = \langle \rangle$:

  \[
  \langle \rangle \in \|\delta_1; \delta_2\|_w^z
  \]

  iff $\langle z, \delta_1; \delta_2 \rangle \in \mathcal{F}_w$ \hspace{1cm} (by Definition 5.7)

  iff $\langle z, \delta_1 \rangle \in \mathcal{F}_w$ and $\langle z, \delta_2 \rangle \in \mathcal{F}_w$ \hspace{1cm} (by Definition 5.5)

  iff $\langle \rangle \in \|\delta_1\|_w^z$ and $\langle \rangle \in \|\delta_2\|_w^z$ \hspace{1cm} (by Definition 5.7)

- $z' = p \cdot z''$:

  \[
  p \cdot z'' \in \|\delta_1; \delta_2\|_w^z
  \]

  iff $\langle z, \delta_1; \delta_2 \rangle \xrightarrow{w}(z \cdot p \cdot z'', \delta')$ and $\langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}_w$ \hspace{1cm} (by Definition 5.7)

  iff $\langle z, \delta_1; \delta_2 \rangle \xrightarrow{w}(z \cdot p, \delta''), \langle z \cdot p, \delta'' \rangle \xrightarrow{w}(z \cdot p \cdot z'', \delta')$

  and $\langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}_w$ \hspace{1cm} (by definition of $\xrightarrow{w}$)

  iff $\delta'' = \gamma; \delta_2$ and $\langle z, \delta_1 \rangle \xrightarrow{w}(z \cdot p, \gamma)$ or $\langle z, \delta_1 \rangle \in \mathcal{F}_w$ and $\langle z, \delta_2 \rangle \xrightarrow{w}(z \cdot p, \delta'' \rangle$,

  \[
  \langle z \cdot p, \delta'' \rangle \xrightarrow{w}(z \cdot p \cdot z'', \delta') \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}_w
  \] \hspace{1cm} (by Definition 5.5)
5.1 The Logic \( \mathcal{LSG} \)

4. This item can be shown by distinguishing two cases:

- **\( z' = \langle \rangle \):**

  \[
  \langle \rangle \in |\delta_1 \delta_2|_w
  \]

  \[
  \text{iff } \langle z, \delta_1 \delta_2 \rangle \in F^w \quad (\text{by Definition 5.7})
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \in F^w \text{ or } \langle z, \delta_2 \rangle \in F^w \quad (\text{by Definition 5.5})
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \in F^w \text{ or } \langle z, \delta_2 \rangle \in F^w \quad (\text{by Definition 5.7})
  \]

- **\( z' = p \cdot z'' \):**

  \[
  p \cdot z'' \in |\delta_1 \delta_2|_w
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \xrightarrow{w} \langle z, \delta_2 \rangle \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in F^w \quad (\text{by Definition 5.7})
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \xrightarrow{w} \langle z \cdot p, \delta'' \rangle, \langle z \cdot p, \delta'' \rangle \xrightarrow{w^*} \langle z \cdot p \cdot z'', \delta' \rangle
  \]

  \[
  \text{and } \langle z \cdot p \cdot z'', \delta' \rangle \in F^w \quad (\text{by definition of } \xrightarrow{w^*})
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \xrightarrow{w} \langle z \cdot p, \delta'' \rangle \text{ or } \langle z, \delta_2 \rangle \xrightarrow{w} \langle z \cdot p, \delta'' \rangle,\]

  \[
  \langle z \cdot p, \delta'' \rangle \xrightarrow{w^*} \langle z \cdot p \cdot z'', \delta' \rangle \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in F^w \quad (\text{by Definition 5.5})
  \]

  \[
  \text{iff } \langle z, \delta_1 \rangle \xrightarrow{w^*} \langle z \cdot p \cdot z'', \delta' \rangle \text{ or } \langle z, \delta_2 \rangle \xrightarrow{w^*} \langle z \cdot p \cdot z'', \delta' \rangle
  \]

  \[
  \text{and } \langle z \cdot p \cdot z'', \delta' \rangle \in F^w \quad (\text{by definition of } \xrightarrow{w^*})
  \]

  \[
  \text{iff } p \cdot z'' \in |\delta_1|_w \text{ or } p \cdot z'' \in |\delta_2|_w
  \]

  \[
  \text{iff } p \cdot z'' \in |\delta_1|_w \cup |\delta_2|_w \quad (\text{by Definition 5.7})
  \]

5. For this item we make a similar case distinction as above:

- **\( z' = \langle \rangle \):**

  \[
  \langle \rangle \in |\pi x. \delta|_w
  \]

  \[
  \text{iff } \langle z, \pi x. \delta \rangle \in F^w \quad (\text{by Definition 5.7})
  \]
We further need the following definition:

\[
\langle z, \delta^n \rangle \in \mathcal{F}^w \text{ for some } n \in \mathcal{N}_x \\
\langle \rangle \in \mathcal{D}^z_n w \text{ for some } n \in \mathcal{N}_x 
\]

\[
\begin{align*}
\bullet \quad z' &= p \cdot z''; \\
p \cdot z'' &\in \mathcal{D}^z_n w \text{ (by Definition 5.5)}
\end{align*}
\]

\[
\begin{align*}
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}^w \text{ (by Definition 5.7)} \\
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ (by Definition 5.7)}
\end{align*}
\]

\[
\begin{align*}
\bullet \quad z' &= \langle \rangle; \text{ Obvious, since } \langle \rangle \in \|z^n\|_w \text{ and } z' = z_0 \text{ with } z_0 = \langle \rangle. \\
\bullet \quad z' &= p \cdot z''; \\
p \cdot z'' &\in \mathcal{D}^z_n w \text{ (by Definition 5.7)}
\end{align*}
\]

\[
\begin{align*}
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}^w \text{ (by Definition 5.7)} \\
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ (by Definition 5.7)}
\end{align*}
\]

\[
\begin{align*}
6. \text{ For this item, we make again an induction on the length of } z': \\
\bullet \quad z' &= \langle \rangle; \text{ Obvious, since } \langle \rangle \in \|z^n\|_w \text{ and } z' = z_0 \text{ with } z_0 = \langle \rangle. \\
\bullet \quad z' &= p \cdot z''; \\
p \cdot z'' &\in \mathcal{D}^z_n w \text{ (by Definition 5.7)}
\end{align*}
\]

\[
\begin{align*}
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ and } \langle z \cdot p \cdot z'', \delta' \rangle \in \mathcal{F}^w \text{ (by Definition 5.7)} \\
\text{iff } \langle z, \delta^n \rangle &\in \mathcal{F}^w \text{ (by Definition 5.7)}
\end{align*}
\]

We further need the following definition:
**Definition 5.11** (Precondition Extension). Let $\delta$ be a program. Then $\delta^p$, called the *precondition extension of $\delta$*, denotes the result of replacing every atomic action $t$ in $\delta$ by $\text{Poss}(t)?; t$.

The above is necessary because according to $\text{Do}$, an atomic action action can only be executed in case its precondition holds, while this is not required in the transition semantics of Definition 5.5.3 We then have:

**Theorem 5.12.** Let $\delta$ be a simple Golog program. Then

$$\models \text{Do}(\delta, \alpha) \equiv \langle \delta^p \rangle \alpha.$$  

*Proof.* We prove $w, z \models \text{Do}(\delta, \alpha)$ iff $w, z \models (\delta^p) \alpha$ by structural induction on $\delta$:

- $\delta = t$:
  
  $w, z \models \text{Do}(t, \alpha)$
  
  iff $w, z \models \text{Poss}(t) \land [t] \alpha$  
  (by Definition 3.56)
  
  iff $w, z \models \text{Poss}(t)$ and $w, z \cdot p \models \alpha$, where $p = |t|^z_w$  
  (by the semantics)
  
  iff $|\text{Poss}(t)??|^z_w = \{\langle\rangle\}$ and $|t|^z_w = \{p\}$ and $w, z \cdot p \models \alpha$, where $p = |t|^z_w$  
  (by Lemma 5.10 items 1 and 2)
  
  iff $|\text{Poss}(t)?; t|^z_w = \{p\}$ and $w, z \cdot p \models \alpha$, where $p = |t|^z_w$  
  (by Lemma 5.10 item 3)
  
  iff for some $z' \in |\text{Poss}(t)?; t|^z_w$, $w, z \cdot z' \models \alpha$  
  (by the semantics)
  
  iff $w, z \models \langle \text{Poss}(t)?; t \rangle \alpha$  
  (by the semantics)
  
- $\delta = \phi?$
  
  $w, z \models \text{Do}(\phi?; \alpha)$
  
  iff $w, z \models \phi \land \alpha$  
  (by Definition 3.56)
  
  iff $w, z \models \phi$ and $w, z \models \alpha$  
  (by the semantics)
  
  iff $|\phi??|^z_w = \{\langle\rangle\}$ and $w, z \models \alpha$  
  (by Lemma 5.10 item 2)
  
  iff for some $z' \in |\phi??|^z_w$, $w, z \cdot z' \models \alpha$  
  (by the semantics)
  
  iff $w, z \models \langle \phi?? \rangle \alpha$  
  (since $\phi?? = \phi?$)

3Of course we could define an alternative version of the transition semantics that also includes preconditions, but then the meaning of the $[t]$ operator in case of atomic actions would differ from the one it has in $\mathcal{ES}$.  

\[ \delta = \delta_1; \delta_2: \]
\[ w, z \models Do(\delta_1; \delta_2, \alpha) \]
\[ \text{iff } w, z \models Do(\delta_1, Do(\delta_2, \alpha)) \] (by Definition 3.56)
\[ \text{iff } w, z \models \langle \delta_1^{p} \rangle Do(\delta_2, \alpha) \] (by induction)
\[ \text{iff for some } z' \in [\delta_1^{p}]^z_w, w, z \cdot z' \models Do(\delta_2, \alpha) \] (by the semantics)
\[ \text{iff for some } z' \in [\delta_1^{p}]^z_w, w, z \cdot z' \models \langle \delta_2^{p} \rangle \alpha \] (by induction)
\[ \text{iff for some } z' \in [\delta_1^{p}]^z_w \text{ and some } z'' \in [\delta_2^{p}]^z_w, w, z \cdot z' \cdot z'' \models \alpha \] (by the semantics)
\[ \text{iff for some } z'' \in [\delta_1^{p}; \delta_2^{p}]^z_w, w, z \cdot z'' \models \alpha \] (by Lemma 5.10 item 3)
\[ \text{iff } w, z \models \langle \delta_1^{p}; \delta_2^{p} \rangle \alpha \] (by the semantics)
\[ \text{iff } w, z \models \langle (\delta_1; \delta_2)^{p} \rangle \alpha \] (since \( \delta_1^{p}; \delta_2^{p} = (\delta_1; \delta_2)^{p} \))

\[ \delta = \delta_1 | \delta_2: \]
\[ w, z \models Do(\delta_1 | \delta_2, \alpha) \]
\[ \text{iff } w, z \models Do(\delta_1, \alpha) \lor Do(\delta_2, \alpha) \] (by Definition 3.56)
\[ \text{iff } w, z \models Do(\delta_1, \alpha) \text{ or } w, z \models Do(\delta_2, \alpha) \] (by the semantics)
\[ \text{iff } w, z \models \langle \delta_1^{p} \rangle \alpha \text{ or } w, z \models \langle \delta_2^{p} \rangle \alpha \] (by induction)
\[ \text{iff for some } z' \in [\delta_1^{p}]^z_w \text{ or some } z' \in [\delta_2^{p}]^z_w, w, z \cdot z' \models \alpha \] (by the semantics)
\[ \text{iff for some } z' \in [\delta_1^{p}]^z_w \cup [\delta_2^{p}]^z_w, w, z \cdot z' \models \alpha \] (by set theory)
\[ \text{iff for some } z' \in [\delta_1^{p}; \delta_2^{p}]^z_w, w, z \cdot z' \models \alpha \] (by Lemma 5.10 item 4)
\[ \text{iff } w, z \models \langle \delta_1^{p}; \delta_2^{p} \rangle \alpha \] (by the semantics)
\[ \text{iff } w, z \models \langle (\delta_1 | \delta_2)^{p} \rangle \alpha \] (since \( \delta_1^{p}; \delta_2^{p} = (\delta_1 | \delta_2)^{p} \))

\[ \delta = \pi x. \delta_1: \]
\[ w, z \models Do(\pi x. \delta_1, \alpha) \]
\[ \text{iff } w, z \models \exists x. Do(\delta_1, \alpha) \] (by Definition 3.56)
\[ \text{iff for some } n \in N_x, w, z \models Do(\delta_1, \alpha)^x_n \] (by the semantics)
\[ \text{iff for some } n \in N_x, w, z \models Do(\delta_1^x, \alpha) \] (see below)
\[ \text{iff for some } n \in N_x, w, z \models \langle \delta_1^x \rangle \alpha \] (by induction)
\[ \text{iff for some } n \in N_x \text{ and some } z' \in [\delta_1^x]^z_w, w, z \cdot z' \models \alpha \] (by the semantics)
\[ \text{iff for some } n \in N_x \text{ and some } z' \in [\langle \delta_1^x \rangle]^z_w, w, z \cdot z' \models \alpha \] (since \( \delta_1^x = (\delta_1^x)^x_n \))
\[ \text{iff for some } z' \in [\pi x. \delta_1]^z_w, w, z \cdot z' \models \alpha \] (by Lemma 5.10 item 5)
5.1 The Logic ESG

iff \( w, z \models (\pi x. \delta_1^p) \alpha \)  
iff \( w, z \models ((\pi x. \delta_1) p) \alpha \)  
(by the semantics)  

Above we used that \( Do(\delta_1, \alpha)_n^z = Do(\delta_1^p, \alpha) \), which can be shown by a simple induction on \( \delta \), where we assume without loss of generality that \( x \) does not appear freely in \( \alpha \).

• \( \delta = \delta_1^* \):

\( \Rightarrow \): Let \( w, z \models Do(\delta_1^*, \alpha) \). Therefore for all \( w' \) with \( w' \not\sim_P w \), if \( w', z \models \Box(\alpha \supset P) \) and \( w', z \models \Box(Do(\delta_1, P) \supset P) \), then \( w'[P, z] = 1 \). Now let \( w_0 \) be a world such that for all \( z' \),

\[
(5.15) \quad w_0[P, z \cdot z'] = 1 \quad \text{iff there is some } z'' \in \| (\delta_1^*)^p \|_w^z \text{ with } w, z \cdot z' \cdot z'' \models \alpha
\]

and which is otherwise like \( w \). Clearly, \( w_0 \not\sim_P w \). We will now show that

(i) \( w_0, z \models \Box(\alpha \supset P) \) and

(ii) \( w_0, z \models \Box(Do(\delta_1, P) \supset P) \).

From this it follows that \( w_0[P, z] = 1 \), and hence by (5.15) that there is some \( z'' \in \| (\delta_1^*)^p \|_w^z \) with \( w, z \cdot z' \cdot z'' \models \alpha \), therefore \( w, z \models ( (\delta_1^*)^p ) \alpha \).

To prove (i), let \( w_0, z \cdot z' \models \alpha \). Recall that it is assumed that \( \alpha \) does not contain \( P \), hence it is easy to show (by an induction on the structure of \( \alpha \)) that also \( w, z \cdot z' \models \alpha \). Since \( \langle \rangle \in \| \delta_1^p \|_w^z \) for any \( z' \), we have \( w_0[P, z \cdot z'] = 1 \).

For (ii), let \( w_0, z \cdot z' \models Do(\delta_1, P) \). By induction, \( w_0, z \cdot z' \models (\delta_1^p) P \). Semantically this means that there is some \( z'' \in \| \delta_1^p \|_w^z \) such that \( w_0[P, z \cdot z' \cdot z''] = 1 \). Since \( P \) is assumed to not occur freely in \( \delta_1 \), it is also easy to show that \( z'' \in \| \delta_1^p \|_w^z \). By (5.15), there is some \( z''' \in \| (\delta_1^*)^p \|_w^z \) with \( w, z \cdot z' \cdot z'' \cdot z''' \models \alpha \). Lemma 5.10 item 6 implies that \( z''' \) has the form \( z_0 \cdot z_1 \cdots z_k \), where \( z_0 = \langle \rangle \) and \( z_{i+1} \in \| \delta_1^p \|_w^z \cdot z'' \cdot z'' \cdot z'' \). Using Lemma 5.10 item 6 again, and since \( z'' \in \| \delta_1^p \|_w^z \), we obtain that \( z' \cdot z'' \in \| (\delta_1^*)^p \|_w^z \) as well. Therefore, \( w_0[P, z \cdot z'] = 1 \).

\( \Leftarrow \): Let \( w, z \models ( (\delta_1^*)^p ) \alpha \) and for some \( w' \) with \( w' \not\sim_P w \), let

(i) \( w', z \models \Box(\alpha \supset P) \) and

(ii) \( w', z \models \Box(Do(\delta_1, P) \supset P) \).

We have to show that \( w'[P, z] = 1 \). By induction, \( Do(\delta_1, P) \) is equivalent to \((\delta_1^p) \alpha \), therefore (ii) holds just in case that

(iii) for all \( z'' \), if \( w'[P, z \cdot z'' \cdot z'''] = 1 \) for some \( z''' \in \| \delta_1^p \|_w^z \), then also \( w'[P, z \cdot z'''] = 1 \).
By assumption, there is some $z' \in \| (\delta_1^*)^p \|^w_w$ and $w, z \cdot z' \models \alpha$. By Lemma 5.10 item 6, $z' = z_0 \cdot z_1 \cdots z_k$, where $z_0 = ()$ and $z_{i+1} \in \| \delta_1^p \|^z_{w \cdot z_0 \cdots z_i}$. Using (i) and the (again easily provable) fact that $w, z \cdot z' \models \alpha$ iff $w', z \cdot z' \models \alpha$, we obtain $w'[P, z \cdot z'] = 1$. Applying (iii) repeatedly, we get $w'[P, z_0 \cdots z_i] = 1$ for all $0 \leq i < k$. Thus in particular $w'[P, z] = 1$. 

Note that with the above theorem we also indirectly established the relation of our new transition semantics to the original Situation Calculus definition of GOLOG [LRL+97] since it was shown in [LL05a] that the ES Do is (under certain assumptions) equivalent to the Situation Calculus Do. On the other hand, there is a discrepancy between our transition semantics and the one provided for the original ConGolog of De Giacomo et al. [GLL00]: Whereas they treat tests as an additional form of transitions (that leave the current situation unchanged), we regard them here as mere conditions (e.g. for program termination, for branching etc) and require that every transition actually corresponds to a physical action.

The difference between the two lies in the question of when during the execution of a concurrent program $\delta_1 | \delta_2$ it is possible to switch from one subprocess to the other. The semantics of De Giacomo et al. allows executions which switch over from one process to the other only to do a single test transition, and immediately switch back again to the first process. Sometimes, this behaviour may be desired, e.g. when a test $\phi$ plays the role of a “wait for” action that is supposed to block a process until a certain condition becomes true. Often however this needs to be avoided, in particular in the case of conditionals and while loops. For instance, when we have a conditional statement

\[
\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endIf},
\]

it does not seem to make much sense to allow an execution in which $\phi$ is first successfully tested to be true, after which a switch to a concurrent process occurs, that process turns $\phi$ to false, and finally execution continues with the $\delta_1$ branch. For this reason, De Giacomo et al. introduce synchronized versions of if and while that ensure that once the decision of which branch to chose or whether to enter the body of the loop has been made, at least one physical action is performed before program execution switches to a concurrent process again. The advantage of not having tests as transitions, as is done in this thesis, is that we get synchronization “for free” and can thus stick with the simple definitions of if-then-else and while as given in (5.5) and (5.6), respectively.
5.1 The Logic $\mathcal{ESG}$

5.1.4 The Coffee Robot Example

Throughout this chapter, we will keep referring to our running example of the coffee delivery robot. Below is the definition of the corresponding basic action theory. BATs in $\mathcal{ESG}$ are no different from those we formulate in $\mathcal{ES}$, and hence also all related results, in particular regarding regression, are still valid.

The actions at the robot’s disposal are assumed to be $\text{wait}$ (do nothing for a while), $\text{selectRequest}(x)$ (select $x$’s request to be served next), $\text{pickupCoffee}$ (grab a cup of coffee) and $\text{bringCoffee}(x)$ (bring a cup of coffee to $x$). Additionally, there is an exogenous action $\text{requestCoffee}(x)$ that represents people sending coffee requests to the robot. The main fluents in this domain are $\text{queue}$ (the current queue of requests, functional) and $\text{HoldingCoffee}$ (relational). Sometimes we will also abbreviate $\text{selectRequest}$, $\text{requestCoffee}$, $\text{pickupCoffee}$, $\text{bringCoffee}$ and $\text{HoldingCoffee}$ by $sR$, $rC$, $pC$, $bC$ and $HC$, respectively.

Queues

As mentioned earlier, we imagine that the robot keeps an internal queue of the currently pending coffee requests. Deviating from the original formulation in [GTR97], we do not axiomatize the queue using second-order inductive definitions, but for the sake of simplicity resort to a $k$-ary function symbol $\text{list}$, where $k$ is the maximal size of the queue. The arguments of $\text{list}$ then denote the requests stored in the queue, where a distinguished standard name $e$ is used to represent an empty slot. If $k = 3$, then $\text{list}(\text{Ann}, \text{Bob}, e)$ for example encodes the queue where Ann’s request is at the top, Bob’s is second and no third request has been added yet.

We require in this case that queues satisfy the unique names assumption. We can achieve this by putting the following into the initial theory $\Sigma_0$:

\begin{equation}
\text{list}(x_1, \ldots, x_k) = \text{list}(y_1, \ldots, y_k) \supset x_1 = y_1 \land \cdots \land x_k = y_k
\end{equation}

Since by default all $\text{object}$ function symbols were defined to be fluent, we also include the following successor state axiom in $\Sigma_{\text{init}}$:

\begin{equation}
\Box[a] \text{list}(x_1, \ldots, x_k) = y \equiv \text{list}(x_1, \ldots, x_k) = y
\end{equation}

Consequently, (5.16) will then hold in all situations.

As a more convenient notation we will often write $\langle x_1, \ldots, x_k \rangle$ instead of $\text{list}(x_1, \ldots, x_k)$, but it should be kept in mind that queues are really only ordinary terms when it comes to their semantical interpretation, regression and the like.
We will furthermore use some abbreviations related to queues to express when an item is the first element of a queue, a queue is empty or full, a queue has at least one free slot, or what is the resulting queue when an element is added or removed, respectively:

\[ \text{IsFirst}(q, x) \overset{\text{def}}{=} \exists x_2 \ldots \exists x_k. q = \langle x, x_2, \ldots, x_k \rangle \]

\[ \text{Empty}(q) \overset{\text{def}}{=} q = \langle e, \ldots, e \rangle \]

\[ \text{Full}(q) \overset{\text{def}}{=} \exists x_1 \ldots \exists x_k. q = \langle x_1, \ldots, x_k \rangle \land \bigwedge_{i=1}^{k} x_i \neq e \]

\[ \text{LastFree}(q) \overset{\text{def}}{=} \exists x_1 \ldots \exists x_{k-1}. q = \langle x_1, \ldots, x_{k-1}, e \rangle \]

\[ \text{Enqueue}(q_0, x, q_n) \overset{\text{def}}{=} \bigvee_{i=0}^{k-1} \exists x_1 \ldots \exists x_i. \bigwedge_{j=1}^{i} x_j \neq e \land q_0 = \langle x_1, \ldots, x_i, e, \ldots, e \rangle \land q_n = \langle x_1, \ldots, x_i, x, e, \ldots, e \rangle \]

\[ \text{Dequeue}(q_0, x, q_n) \overset{\text{def}}{=} \exists x_2 \ldots \exists x_k. q_0 = \langle x, x_2, \ldots, x_k \rangle \land q_n = \langle x_2, \ldots, x_k, e \rangle \]

### 5.1.5 Control Program

For the most part we will assume that the robot is controlled by the precondition-extended variant of program (5.1), presented below:

```
loop
  if \neg \text{Empty}(\text{queue})
    then (\pi x)
      Poss(selectRequest(x))?; selectRequest(x);
      Poss(pickupCoffee)?; pickupCoffee;
      Poss(bringCoffee(x))?; bringCoffee(x)
    else Poss(wait)?; wait
  endIf
endLoop
```

### Exogenous Actions

Exogenous actions are those which are not under the control of the agent, but can happen at any time during the runtime of the robot. In our scenario, there is only one such action, namely \text{requestCoffee}(x), representing a newly arriving coffee request by person \( x \). To distinguish exogenous from ordinary actions, we let the definitional part \( \Sigma_{\text{def}} \) of our BAT contain the
5.1 The Logic $\mathcal{ESG}$

**Axiom**

\[ (5.25) \quad \square \text{Exo}(a) \equiv \exists x. \ a = \text{requestCoffee}(x) \]

Let $\delta$ be the control program of the robot. In addition, we have a program

\[ (5.26) \quad \delta_{\text{exo}} \overset{\text{def}}{=} (\pi a. \ \text{Exo}(a)?; a)^\omega \]

that continuously executes exogenous actions at random. The interaction between robot behaviour and its environment is then modelled by the program

\[ \delta \parallel \delta_{\text{exo}}. \]

**Occurrence of Actions**

Sometimes we need to be able to refer to the previously executed action, for example in temporal properties such as (5.2) and (5.3). We can do this by means of the $\text{Occ}$ predicate, which we can treat as any other normal fluent by including

\[ (5.27) \quad \forall a. \ \neg \text{Occ}(a) \]

into $\Sigma_0$ and

\[ (5.28) \quad \square [a] \text{Occ}(a') \equiv (a = a') \]

into $\Sigma_{\text{post}}$.

**Preconditions**

Moreover, we include a precondition axiom in $\Sigma_{\text{def}}$ as follows:

\[ (5.29) \quad \square \text{Poss}(a) \equiv \]

\[ (a = \text{wait}) \lor \]

\[ \exists x \ (a = \text{requestCoffee}(x) \land x \neq e \land \text{LastFree(queue)}) \lor \]

\[ \exists x \ (a = \text{selectRequest}(x) \land x \neq e \land \text{IsFirst(queue, x)}) \lor \]

\[ (a = \text{pickupCoffee} \land \neg \text{HoldingCoffee}) \lor \]

\[ \exists x \ (a = \text{bringCoffee}(x) \land \text{HoldingCoffee}) \]
Successor State Axioms

Finally, we have the following successor state axioms for the queue and HoldingCoffee fluents in $\Sigma_{\text{post}}$:

\begin{align}
(5.30) \quad \Box[a] \text{HoldingCoffee} &\equiv \\
& a = \text{pickupCoffee} \lor \text{HoldingCoffee} \land \neg \exists x. a = \text{bringCoffee}(x) \\
\end{align}

\begin{align}
(5.31) \quad \Box[a] \text{queue} = y &\equiv \\
& \exists x \ (a = \text{requestCoffee}(x) \land \text{Enqueue(queue, } x, y)) \lor \\
& \exists x \ (a = \text{selectRequest}(x) \land \text{Dequeue(queue, } x, y)) \lor \\
& \text{queue} = y \land \neg \exists x (a = \text{requestCoffee}(x) \lor a = \text{selectRequest}(x))
\end{align}

5.2 Characteristic Graphs

Next, we introduce a graph-based representation for program traces and show that any GOLOG program admitted by (5.4) can be represented in this manner.

5.2.1 Trace Graphs

Definition 5.13 (Trace Graphs). A trace graph is given by a triple $\mathcal{G} = \langle V, E, v_0 \rangle$, where

- $V$ is a finite set of vertices such that each vertex $v \in V$ is associated with a situation formula $\varphi(v)$;
- $E$ is a finite set of edges of the form $v \xrightarrow{\psi_1/\pi\bar{x}/\psi_2} v'$, where $v, v' \in V$, $\bar{x}$ is a possibly empty list of variables, $t$ is an action term and $\psi_1, \psi_2$ are situation formulas;
- $v_0 \in V$ is a distinguished vertex called the initial node.

As notational convention, we omit the leading $\pi$ in edges in case the $\bar{x}$ is empty, and we omit any $\psi_i$ that is $\top$.

The intuition behind trace graphs is that nodes represent possible system configurations, whereas edges encode possible transitions among them. For that matter, each node contains a termination condition $\varphi(v)$, while edges are labelled with a list of variables $\bar{x}$ to be (newly) instantiated, the action $t$ to be taken, as well as a first transition condition $\psi_1$ (to be tested prior to (re-) instantiating the $\bar{x}$) and a second transition condition $\psi_2$ (to be tested after (re-) instantiating the $\bar{x}$). Similar to program traces, there are both finite and infinite traces admitted by trace graphs. The former correspond to finite paths through the graph that start in the initial node,
along which all encountered transition conditions are satisfied (wrt a given initial $w$ and $z$) and that ends in a node whose termination condition is satisfied as well. An infinite trace furthermore is given by an infinite path starting in $v_0$ such that no visited node’s termination condition is ever satisfied. To formalize this intuition, we first need the notion of variable maps:

**Definition 5.14** (Variable Maps). A variable map $\theta$ is an assignment that maps each variable to a standard name of the appropriate sort. We write $\theta[x/n]$ to denote the variable map that is like $\theta$, except that it maps $x$ to $n$. When $\vec{x}$ abbreviates $x_1, \ldots, x_k$ and $\vec{n}$ stands for $n_1, \ldots, n_k$, we use $\theta[\vec{x}/\vec{n}]$ as an abbreviation for $\theta[x_1/n_1] \cdots [x_k/n_k]$.

For any formula $\phi$, we let $\phi\theta$ stand for the application of $\theta$ to $\phi$, which refers to the result of substituting each free variable $x$ in $\phi$ by $\theta(x)$, and similarly for terms and programs.

One consequence of the definition we will frequently use is that for any formula $\phi$, variable map $\theta$, variable $y$ and standard name $n$ of the same sort as $y$,

\[
\phi\theta[y/n] = \phi^n_y \theta.
\]

Again, the same is true for terms and programs. Without loss of generality, we will from now on also assume that all variables, no matter whether free or quantified by $\pi$, $\exists$ or $\forall$, are distinct.

**Definition 5.15** (Traces Admitted by Trace Graphs). Given a trace graph $G = (V, E, v_0)$, a graph configuration is of the form $\langle v, \theta, w, z \rangle$ where $v \in V$, $\theta$ is a variable map, $w \in W$ and $z \in Z$. Similar to program execution, we define transition steps and finality of graph configurations as follows:

- $\langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta', w, z \cdot p \rangle$ iff there is an edge $v \xrightarrow{\psi_1/x\psi_2} v' \in E$ and some standard names $\vec{n}$ such that $\theta' = \theta[\vec{x}/\vec{n}]$, $p = |t\theta'|^z_w$, and $w, z \models \psi_1\theta \land \psi_2\theta'$.

- $\langle v, \theta, w, z \rangle \in \mathcal{F}G$ iff $w, z \models \varphi(v)\theta$.

Let $w \in W$, $z \in Z$ and $\theta$ be a variable map. There are both finite and infinite runs through a trace graph:

1. A finite run $r$ from $z$ wrt $w$ and $\theta_0$ through $G$ is of the form

   \[
   \langle v_0, \theta_0, w, z \rangle \xrightarrow{G} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G} \cdots \xrightarrow{G} \langle v_k, \theta_k, w, z \cdot p_1 \cdot p_2 \cdots p_k \rangle
   \]

   where $k \geq 0$, $v_0$ is the initial node of $G$ and $\langle v_k, \theta_k, w, z \cdot p_1 \cdot p_2 \cdots p_k \rangle \in \mathcal{F}G$. The trace of $r$ then is

   \[
   |r| = p_1 \cdot p_2 \cdots p_k
   \]
2. An infinite run \( r \) from \( z \) wrt \( w \) and \( \theta_0 \) through \( \mathcal{G} \) is an infinite sequence \(^4\)

\[
(v_0, \theta_0, w, z) \xrightarrow{\mathcal{G}} (v_1, \theta_1, w, z \cdot p_1) \xrightarrow{\mathcal{G}} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \xrightarrow{\mathcal{G}} (v_3, \theta_3, w, z \cdot p_1 \cdot p_2 \cdot p_3) \xrightarrow{\mathcal{G}} \cdots
\]

where \( v_0 \) is the initial node of \( \mathcal{G} \) and for all \( i \geq 0 \), \( (v_i, \theta_i, w, z \cdot p_1 \cdots p_i) \not\in \mathcal{F}^\mathcal{G} \). The trace of \( r \) then is

\[
\|r\| = p_1 \cdot p_2 \cdot p_3 \cdots
\]

The set of all traces admitted by \( \mathcal{G} \), given \( w, z \) and \( \theta \) is

\[
\|\mathcal{G}\|^{z}_{w}(\theta) = \{\|r\| \mid r \text{ is a (finite or infinite) run from } z \text{ wrt } w \text{ and } \theta \text{ through } \mathcal{G}\}.
\]

5.2.2 Characteristic Program Graphs

It will now become apparent why we restricted ourselves in Section 5.1.1 to the class GOLO (programs admitted by (5.4)): All programs within this class are representable by means of a trace graph. Intuitively, this is due to the fact that given any such program \( \delta \), the set of reachable subprograms (modulo substitution of free variables) is finite.

**Definition 5.16** (Characteristic Program Graphs). Let \( \delta \) be a program. The characteristic graph of \( \delta \) is given by a trace graph \( \mathcal{G}_\delta = (V, E, v_0) \), where vertices \( v \in V \) have the form \( \langle \delta', \phi \rangle \). Intuitively, each such node represents the set of all program configurations \( \langle z, \delta' \rangle \) with the same remaining subprogram \( \delta' \), while the edges encode the transition relation among those configurations. The second component \( \phi \) furthermore indicates whether \( \delta' \) is final, i.e. \( \phi = \phi(v) \) is the termination condition in the terminology of Definition 5.13. \( \mathcal{G}_\delta \) is defined by induction on the structure of \( \delta \):

- \( \delta \) is a primitive action \( t \):
  - In this case, we have two nodes, connected by a single edge labelled with \( t \), where the initial is not final, but the second one is:
    \[
    v_0 = \langle t, \bot \rangle;
    \]
    \[
    V = \{ \langle t, \bot \rangle, \langle nil, \top \rangle \};
    \]
    \[
    E = \{ \langle t, \bot \rangle \xrightarrow{t} \langle nil, \top \rangle \}.
    \]
  - The graph is depicted in Figure 5.1(a).

\(^4\)An infinite run is again formally defined as a mapping from the natural numbers to graph configurations such that each subsequent configuration is reachable by a transition step from its predecessor.
Figure 5.1: Example characteristic graphs
• $\delta$ is a test $\alpha$?
Here we have a single node with $\alpha$? as its program and $\alpha$ as termination condition:

$v_0 = (\alpha?, \alpha)$;

$V = \{\langle \alpha?, \alpha \rangle \}$;

$E = \emptyset$.

The corresponding graph is shown in Figure 5.1(b).

• $\delta$ is a sequence of programs $\delta_1; \delta_2$:
Let $\mathcal{G}_{\delta_1} = (V_1, E_1, v_0^1)$ and $\mathcal{G}_{\delta_2} = (V_2, E_2, v_0^2)$, where $v_0^1 = (\delta_1, \varphi_0^1)$. Then

$v_0 = (\delta_1; \delta_2, \varphi_0^1 \wedge \varphi_0^2)$;

$V = \{\langle \delta_1; \delta_2, \varphi_0^1 \wedge \varphi_0^2 \rangle \} \cup V_1 \cup V_2$;

$E = \{\langle \delta_1; \delta_2, \varphi_0^1 \wedge \varphi_0^2 \rangle \overset{\phi_1/\pi \mathcal{X}_1/\phi_1'}{\rightarrow} \langle \delta_1''; \delta_2, \varphi_1'' \wedge \varphi_0^2 \rangle \mid \langle \delta_1', \varphi_1' \rangle \overset{\phi_1/\pi \mathcal{X}_1/\phi_1'}{\rightarrow} \langle \delta_1'', \varphi_1'' \rangle \in E_1 \}$ \cup

\[\{\langle \delta_1; \delta_2, \varphi_0^1 \wedge \varphi_0^2 \rangle \overset{\phi_2/\pi \mathcal{X}_2/\phi_2'}{\rightarrow} \langle \delta_2', \varphi_2' \rangle \mid \langle \delta_1', \varphi_1' \rangle \in V_1, \varphi_1' \neq \perp, \langle \delta_2', \varphi_2' \rangle \in E_2 \} \cup \]

$E_2$.

The idea here is to essentially leave $\mathcal{G}_{\delta_1}$ as it is, but where each $\delta_1'$ turns into $\delta_1'; \delta_2$ and each termination condition is augmented by $\varphi_0^2$, the condition under which $\delta_2$ is final. $\mathcal{G}_{\delta_2}$ then remains unchanged and we have copies of edges originally going out of $v_0^2$ also at all $\delta_1'; \delta_2$ nodes, with the additional constraint that $\delta_1'$ is final, i.e. that $\varphi_1'$ holds.

An example graph for a program containing a sequence is given in Figure 5.1(c).

• $\delta$ is a nondeterministic choice between two subprograms $\delta_1|\delta_2$:
Let $\mathcal{G}_{\delta_1} = (V_1, E_1, v_0^1)$ and $\mathcal{G}_{\delta_2} = (V_2, E_2, v_0^2)$, where $v_0^1 = (\delta_1, \varphi_0^1)$. Then

$v_0 = (\delta_1|\delta_2, \varphi_0^1 \vee \varphi_0^2)$;

$V = \{v_0\} \cup V_1 \cup V_2$;

$E = \{v_0 \overset{\pi \mathcal{X}_1/\delta_1'}{\rightarrow} v_i \mid v_0^1 \overset{\pi \mathcal{X}_1/\delta_1'}{\rightarrow} v_i \in E_1 \} \cup E_1 \cup E_2$.

Here we have a copy of each $\mathcal{G}_{\delta_i}$. From the new initial node, there are copies of the outgoing edges of both $v_0^1$. Thus, the first transition corresponds to the commitment to either $\delta_1$ or $\delta_2$, and all subsequent transitions have to be in the chosen subprogram.

A small example of a characteristic graph for a program containing a nondeterministic choice is shown in Figure 5.1(e).
5.2 Characteristic Graphs

• δ is a nondeterministic choice of argument πy.δ₁:
  Let \( G_{δ₁} = (V₁, E₁, v₀) \), where \( v₀ = \langle δ₁, ϕ₀ \rangle \). Then
  \[
  v₀ = \langle πy.δ₁, ∃y.ϕ₀ \rangle;
  \]
  \[
  V = \{ v₀ \} \cup V₁;
  \]
  \[
  E = \{ v₀ \overset{πy.ϕ₁′}{\rightarrow} v′₁ \mid v₀ \overset{πE₁/ϕ₁′}{\rightarrow} v₁ \in E₁ \} \cup E₁.
  \]

In this case we extend a copy of \( G_{δ₁} \) with a new initial node whose outgoing edges contain the additional choice for \( y \) and whose termination condition also requires the existence of an appropriate \( y \).

An example of a simple program with a nondeterministic choice of argument is depicted in Figure 5.1(f).

• δ is the concurrent execution of two subprograms \( δ₁ \mid δ₂ \):
  Let \( G_{δ₁} = (V₁, E₁, v₀) \) and \( G_{δ₂} = (V₂, E₂, v₀) \), where \( v₀ = \langle δ₁, ϕ₀ \rangle \). Then
  \[
  v₀ = \langle δ₁ \parallel δ₂, ϕ₀ \rangle;
  \]
  \[
  V = \{ \langle δ₁′, ϕ₁′ \rangle, \langle δ₂′, ϕ₂′ \rangle \mid \langle δ₁′, ϕ₁′ \rangle ∈ V₁, \langle δ₂′, ϕ₂′ \rangle ∈ V₂ \};
  \]
  \[
  E = \{ \langle δ₁′, ϕ₁′ \rangle \overset{φ₁/πE₁/ϕ₁′}{\rightarrow} \langle δ₁′′, ϕ₁′′ \rangle \mid \langle δ₁′, ϕ₁′ \rangle ∈ V₁, \langle δ₁′′, ϕ₁′′ \rangle ∈ E₁ \} \cup
  \]
  \[
  \{ \langle δ₂′, ϕ₂′ \rangle \overset{φ₂/πE₂/ϕ₂′}{\rightarrow} \langle δ₂′′, ϕ₂′′ \rangle \mid \langle δ₂′, ϕ₂′ \rangle ∈ V₂, \langle δ₂′′, ϕ₂′′ \rangle ∈ E₂ \}.
  \]

The set of vertices here is something like the Cartesian product of the nodes of the \( G_{δ₁} \).

Edges are such that either a transition in \( δ₁ \) or one in \( δ₂ \) is taken, and the other program remains unchanged.

A simple example of the graph for a concurrent execution is provided in Figure 5.1(d).

• δ is an iteration \( ⟨δ₁⟩^* \):
  Let \( G_{δ₁} = (V₁, E₁, v₀) \). Then
  \[
  v₀ = \langle (√δ₁)^*, ⊤ \rangle;
  \]
  \[
  V = \{ v₀ \} \cup \{ \langle δ₁′; (√δ₁)^*, ϕ₁′ \rangle \mid \langle δ₁′, ϕ₁′ \rangle ∈ V₁ \};
  \]
  \[
  E = \{ v₀ \overset{πE₁/ϕ₁′}{\rightarrow} \langle δ₁′; (√δ₁)^*, ϕ₁′ \rangle \mid v₀ \overset{πE₁/ϕ₁′}{\rightarrow} \langle δ₁′, ϕ₁′ \rangle ∈ E₁ \} \cup
  \]
  \[
  \{ \langle δ₁′; (√δ₁)^*, ϕ₁′ \rangle \overset{φ₁/πE₁/ϕ₁′}{\rightarrow} \langle δ₁′′; (√δ₁)^*, ϕ₁′′ \rangle \mid \langle δ₁′, ϕ₁′ \rangle \overset{φ₁/πE₁/ϕ₁′}{\rightarrow} \langle δ₁′′, ϕ₁′′ \rangle ∈ E₁ \} \cup
  \]
  \[
  \{ \langle δ₁′; (√δ₁)^*, ϕ₁′ \rangle \overset{φ₁′/πE₁/ϕ₁′}{\rightarrow} \langle δ₁′′; (√δ₁)^*, ϕ₁′′ \rangle \mid \langle δ₁′, ϕ₁′ \rangle ∈ V₁, \langle δ₁′′, ϕ₁′′ \rangle \neq ⊥ \}
  \]
  \[
  \{ v₀ \overset{πE₁/ϕ₁′}{\rightarrow} \langle δ₁′, ϕ₁′ \rangle \}
  \]
  \[
  \{ v₀ \overset{πE₁/ϕ₁′}{\rightarrow} \langle δ₁′, ϕ₁′ \rangle \}.
  \]
Here we introduce a new initial node, which has $\top$ as the termination condition, and which has copies of the leaving edges of the initial node of $G_{\delta_1}$. Furthermore, we have copies of all edges within $G_{\delta_1}$ where each subprogram $\delta'_1$ simply becomes $\delta'_1; (\delta_1)^*$. Finally, there are edges that encode the case where the $\delta'_1$ is final and a new iteration of $(\delta_1)^*$ is started.

A simple example of the graph for an iteration is depicted in Figure 5.1(g).

A consequence from the above definition is:

**Lemma 5.17.** $G_\delta$ is a trace graph.

**Proof.** This is almost obvious from the definition. One specialty of the definition is that it is assumed that for all edges $v_0 \xrightarrow{\phi_1/\pi \vec{x}; \phi_2} v'$ leaving the initial node $v_0$ of the graph of a subprogram, the corresponding $\phi_1$ is $\top$ (and therefore omitted). It is easy to show by induction on $\delta$ that this property then translates to all programs, and hence that $G_\delta$ is well defined.\(^5\)

Note that in the examples depicted in Figure 5.1 we made use of two conventions with respect to characteristic graphs, which we will keep using throughout the remainder of the thesis: On the one hand, we make “obvious” simplifications for programs and formulas, e.g. both $(\text{nil}; \delta)$ and $(\delta; \text{nil})$ become $\delta$, $\bot \land \alpha$ becomes $\bot$, $\top \lor \beta$ becomes $\top$ etc. We can thus identify “obviously” equivalent nodes and edges, which helps keeping the graphs smaller and simpler.

On the other hand, all nodes and edges are omitted that are unreachable (in the graph-theoretic sense) from the corresponding initial node $v_0$. For example in the graph of Figure 5.1(c) for the program $a|b$, we dropped the nodes $\langle a, \bot \rangle$ and $\langle b, \bot \rangle$ (which originated as the initial nodes of the graphs for the subprograms $a$ and $b$, respectively) as well as their outgoing edges $\langle a, \bot \rangle \xrightarrow{a} \langle \text{nil}, \top \rangle$ and $\langle b, \bot \rangle \xrightarrow{b} \langle \text{nil}, \top \rangle$. This is safe in the sense that no admitted traces are lost since vertices and edges that are unreachable from the initial node can obviously never be part of a successful run through the graph.

Let $\delta_{\text{coffee}}$ denote the control program for the coffee robot given in (5.1). Recall that the exogenous actions of the domain are furthermore represented by the program $\delta_{\text{exo}} = (\pi a. \text{Exo}(a)?; a)^\omega$. Figure 5.2 shows the characteristic graph for the overall program $\delta_{\text{coffee}}|\delta_{\text{exo}}$.

---

\(^5\)In fact it will rarely occur that we get any nontrivial $\phi_1$ in an edge $v \xrightarrow{\phi_1/\pi \vec{x}; \phi_2} v'$. One example is the program $(\tau y.g(y); P(y))?^*$, where when entering the next cycle of the iteration, we have to test for $P(y)$ with the previously chosen instantiation of $y$ before choosing a new instance.
5.2 Characteristic Graphs

where

\[(5.33) \quad v_0 = \langle \delta_{\text{coffee}} \parallel \delta_{\text{exo}}, \bot \rangle \]
\[(5.34) \quad v_1 = \langle (\text{pickupCoffee}; \text{bringCoffee}(x); \delta_{\text{coffee}}) \parallel \delta_{\text{exo}}, \bot \rangle \]
\[(5.35) \quad v_2 = \langle (\text{bringCoffee}(x); \delta_{\text{coffee}}) \parallel \delta_{\text{exo}}, \bot \rangle \]

Intuitively, the graph can be understood as follows: The fact that the program is a nonterminating one is reflected in the cyclic structure of the graph and the termination conditions of all nodes being \(\bot\). In each cycle of the infinite loop, there is the choice between either doing \textit{wait} when the queue is empty (the lower reflexive edge at \(v_0\)) or otherwise executing the sequence \textit{selectRequest}(x); \textit{pickupCoffee}; \textit{bringCoffee}(x) (the cycle between \(v_0\), \(v_1\) and \(v_2\)) after choosing an instantiation for the \(x\). Because exogenous actions can happen at any time, each node furthermore possesses a reflexive edge labelled with \(\pi a : a/\text{Exo}(a)\).

Before we prove that the characteristic graphs correctly capture all program traces, we first need the following lemmas. First, we note that the remaining program in the initial node of a characteristic graph of a program \(\delta\) is \(\delta\) itself:

**Lemma 5.18.** If \(G_\delta = \langle V, E, v_0 \rangle\), then \(v_0 = \langle \delta, \phi \rangle\) for some \(\phi\).

*Proof.* Obvious from Definition 5.16. \(\square\)

Next we have a central lemma that establishes the correspondence between transitions in characteristic graphs and transitions in programs, as well as between finality of graph configurations and finality of programs:
Lemma 5.19. Let δ be a program (possibly with free variables), w ∈ W, and z ∈ Z. Then for all δ′, δ″ appearing in nodes of G(δ) (including δ itself) and all variable maps θ′, θ″:

(1) \langle \langle δ′, \cdot \rangle, θ′, w, z \rangle \xrightarrow{G} \langle \langle δ′′, \cdot \rangle, θ′′, w, z \cdot p \rangle \iff \langle z, δ′θ′ \rangle \xrightarrow{w} \langle z \cdot p, δ′′θ″ \rangle

(2) \langle \langle δ′, \cdot \rangle, θ′, w, z \rangle ∈ F(δ) \iff \langle z, δ′θ′ \rangle ∈ F

Proof. The lemma can be proved by an induction on the structure of δ. The details can be found in Appendix A.2.

Lemma 5.20. Let δ be a program and θ a variable map. If \langle z, δθ \rangle \xrightarrow{w}\ast \langle z′, δ′ \rangle, then δ′ = δ′θ′ for some δ′ appearing in some node of G(δ) and some variable map θ′.

Proof. The proof is by a simple, but tedious outer induction on the structure of δ and an inner induction on the length of z′.

We can now note that characteristic graphs correctly represent programs in the sense that the traces admitted by a program are exactly the traces admitted by its characteristic graph:

Theorem 5.21. Let δ be a program (possibly with free variables), θ a variable map, w ∈ W, and z ∈ Z. Then

\|G(δ)\|_w^z(θ) = |δθ|_w^z

Proof. We prove the two cases of finite and infinite traces separately:

• Finite traces:

z′ ∈ \|G(δ)\|_w^z(θ_0)

iff there is a finite run

\langle \langle δ, \cdot \rangle, θ_0, w, z \rangle \xrightarrow{G} \langle \langle δ_1, \cdot \rangle, θ_1, w, z \cdot p_1 \rangle \xrightarrow{G} \cdots \xrightarrow{G} \langle \langle δ_k, \cdot \rangle, θ_k, w, z \cdot p_1 \cdot p_2 \cdots p_k \rangle

such that z′ = p_1 \cdots p_k, k ≥ 0 and \langle \langle δ_k, \cdot \rangle, θ_k, w, z \cdot p_1 \cdot p_2 \cdots p_k \rangle ∈ F(δ)

(by Definitions 5.15 and 5.16 as well as Lemma 5.18)

iff \langle z, δθ_0 \rangle \xrightarrow{w} \langle z \cdot p_1, δ_1θ_1 \rangle \xrightarrow{w} \cdots \xrightarrow{w} \langle z \cdot p_1 \cdot p_2 \cdots p_k, δ_kθ_k \rangle

such that z′ = p_1 \cdots p_k, k ≥ 0 and \langle z \cdot p_1 \cdot p_2 \cdots p_k, δ_kθ_k \rangle ∈ F

(by Lemma 5.19)

iff \langle z, δθ_0 \rangle \xrightarrow{w}\ast \langle z \cdot z′, δ′ \rangle and \langle z \cdot z′, δ′ \rangle ∈ F

(by Lemma 5.20)

iff z′ ∈ \|δθ_0\|_w^z

(by Definition 5.7)
5.3 Nonterminating Programs

We are now ready to use the characteristic graphs for verifying program properties. The first verification method we discuss for nonterminating programs is inspired by the classic CTL model checking algorithm [CES86]. In a nutshell, the latter operates on a finite transition system which represents a concurrent, nonterminating system. Using an exhaustive search of the state space, the method inductively identifies subsets of states that satisfy the different subformulas of the CTL input formula. Subformulas that include a universal (A) or an existential (E) path quantifier are evaluated by an iterative fixpoint computation.

The algorithm presented in this section evaluates formulas in a similar manner. Consequently, we will consider a CTL-like subset of ESG, as defined in the following subsection. The main differences to classic CTL are that we allow for first-order quantification and that instead of the A and E quantifiers, we use the more general program trace quantifiers \([\delta]\) and \(\langle\delta\rangle\) of ESG. The characteristic graphs then somewhat play the role of the transition system in the sense that they are used to systematically explore the space of possible program configurations.

5.3.1 The Logic ESG\textsubscript{CTL}

**Definition 5.22 (ESG\textsubscript{CTL}).** The formulas of ESG\textsubscript{CTL} are the ones admitted by the following grammar:

\[
\varphi ::= (t_1 = t_2) \mid F(t) \mid \neg \varphi \mid \varphi \land \varphi \mid \exists x. \varphi \mid \langle\delta\rangle X \varphi \mid \langle\delta\rangle (\varphi U \varphi) \mid \langle\delta\rangle G \varphi
\]
In the latter three cases, the program \( \delta \) is required to have the form
\[
\delta_1 \omega \|| \cdots \|| \delta_k \omega.
\] (5.37)

Apart from the fact that we prohibit \([ \cdot ]\) operators here, the major restriction is that we require path operators \( \langle\langle \delta \rangle\rangle \) to be applied directly to subformulas of the form \( X \varphi \), \((\varphi_1 U \varphi_2)\), or \( G \varphi \). In particular, Boolean combinations of temporal subformulas are disallowed. Like in CTL, only using existential path quantifiers is no real restriction, since the following equivalences are valid:

**Lemma 5.23.** Let \( \delta \) be a program that does not admit finite traces. Then:

\[
\begin{align*}
| & = \Box [\delta] X \varphi \quad \equiv \neg \langle\langle \delta \rangle\rangle X \neg \varphi \\
| & = \Box [\delta] G \varphi \quad \equiv \neg \langle\langle \delta \rangle\rangle (\top U \neg \varphi) \\
| & = \Box [\delta] (\varphi_1 U \varphi_2) \quad \equiv \neg \langle\langle \delta \rangle\rangle (\neg \varphi_2 U (\neg \varphi_1 \land \neg \varphi_2)) \land \neg \langle\langle \delta \rangle\rangle G \neg \varphi_2
\end{align*}
\] (5.38) (5.39) (5.40)

**Proof.**

- (5.38) follows directly from our semantics definition:
  \[
  w, z \models [\delta] X \varphi \\
  \text{iff for all } \pi \in [\delta]_w^z, \text{ if } \pi = p \cdot \pi', \text{ then } \pi, w, z \models \varphi \\
  \text{iff there is no } \pi \in [\delta]_w^z \text{ such that } \pi = p \cdot \pi' \text{ and } \pi, w, z \not\models \varphi \\
  \text{iff there is no } \pi \in [\delta]_w^z \text{ such that } \pi, w, z \models X \neg \varphi \\
  \text{iff } w, z \models \neg \langle\langle \delta \rangle\rangle X \neg \varphi
  \]

- (5.39) is also an almost immediate consequence of the semantics:
  \[
  w, z \models [\delta] G \varphi \\
  \text{iff for all } \pi \in [\delta]_w^z \text{ and all } z' \text{ such that } \pi = z' \cdot \pi', \pi \models \varphi \\
  \text{iff there is no } \pi \in [\delta]_w^z \text{ such that for some } z' \text{ with } \pi = z' \cdot \pi', \pi \not\models \varphi \\
  \text{iff there is no } \pi \in [\delta]_w^z \text{ such that } \pi, w, z \models (\top U \neg \varphi) \\
  \text{iff } w, z \models \neg \langle\langle \delta \rangle\rangle (\top U \neg \varphi)
  \]

- We prove the two directions of (5.40) separately:
  \( \Rightarrow \) : Let \( w, z \models [\delta] (\varphi_1 U \varphi_2) \) and assume that \( w, z \models \langle\langle \delta \rangle\rangle (\neg \varphi_2 U (\neg \varphi_1 \land \neg \varphi_1)) \) or \( w, z \models \langle\langle \delta \rangle\rangle G \neg \varphi_2 \). In the former case, there is some \( \pi \in [\delta]_w^z \) such that \( \pi = z' \cdot \pi', \)

\[6\text{We deal with such formulas in Section 5.5.}\]
5.3 Nonterminating Programs

\[ w, z \cdot z', \pi' \models \neg \varphi_1 \land \neg \varphi_2 \text{ and for all } z'' \neq z' \text{ with } z' = z'' \cdot z''' \text{, } w, z \cdot z'' \cdot z''' \cdot \pi' \nmid \varphi_2. \]

Then obviously \( w, z, \pi \nmid (\varphi_1 U \varphi_2) \), contradiction. In the latter case, we get that for some \( \pi \in \| \delta \|_w \), for all \( z' \) with \( \pi = z' \cdot \pi' \), \( w, z \cdot z' \nmid \varphi_2 \). Again, \( w, z, \pi \nmid (\varphi_1 U \varphi_2) \), contradiction.

\( \Leftarrow \): Conversely, let \( w, z \models \neg \langle \delta \rangle (\neg \varphi_2 U (\neg \varphi_1 \land \neg \varphi_2)) \land \neg \langle \delta \rangle G \neg \varphi_2 \) and suppose that \( w, z \nmid \| \delta \|_w (\varphi_1 U \varphi_2) \). This means there is some \( \pi \in \| \delta \|_w \) such that for all \( z' \) with \( \pi = z' \cdot \pi' \), \( w, z \cdot z', \pi' \nmid \varphi_2 \) or there exists \( z'' \neq z' \) with \( z' = z'' \cdot z''' \) so that \( w, z \cdot z'' \cdot z''' \cdot \pi' \nmid \varphi_1 \). Let \( z_0 \) be the smallest action sequence with \( \pi = z_0 \cdot \pi_0 \) and \( w, z \cdot z_0, \pi_0 \models \varphi_2 \). Note that there must be such a sequence as otherwise this would directly contradict \( w, z \models \neg \langle \delta \rangle G \varphi_2 \).

By assumption, there is some \( z_0' \neq z_0 \) such that \( z_0 = z_0' \cdot z_0'' \) and \( w, z \cdot z_0', z_0'' \cdot \pi_0 \nmid \varphi_1 \), and for all \( z_1 \neq z_0 \) with \( z_0 = z_1 \cdot z_1' \), \( w, z_0 \cdot z_1, z_1' \cdot \pi_0' \nmid \varphi_2 \). This however contradicts \( w, z \models \neg \langle \delta \rangle (\neg \varphi_2 U (\neg \varphi_1 \land \neg \varphi_2)) \).

Using the above, example property (5.3) is thus a formula of \( \mathcal{ES} \mathcal{Y}_{\text{CTL}} \), while (5.2) is not,\(^7\) as all traces of programs according to (5.37) are infinite:

**Lemma 5.24.** Let \( \delta \) be a program of the form \( \delta_1^{\omega} \cdots \delta_k^{\omega} \) and \( \theta \) a variable map. Then \( \delta \theta \) does not admit finite traces. Equivalently, if \( G_\delta = (V, E, v_0) \), then for all \( \langle \delta', \varphi' \rangle \in V \), \( | \varphi' \equiv \bot \).

**Proof.** It suffices to show that for all nodes in the characteristic graph of \( \delta \), the termination condition is equivalent to \( \bot \). By Definition 5.15 and Lemma 5.19, there then cannot be any finite trace in \( \| G_\delta \|_w (\theta) \), and hence by Theorem 5.21 neither in \( \| \delta \theta \|_w \).

Let \( G_{\delta_i}^{\omega} = (V_i, E_i, v_0) \) for \( 0 \leq i \leq k \). By the concurrency case of Definition 5.16, all nodes in \( G_\delta \) are of the form \( \langle \delta_1^{\omega_1} \cdots \delta_k^{\omega_k} \varphi_1', \cdots \varphi_k' \rangle \), where \( \langle \delta_1^{\omega_1}, \varphi_1' \rangle \in V_i \). The claim therefore follows if all \( \varphi_i' \) are \( \bot \). It is easy to see that this is indeed the case, as \( \delta_1^{\omega_1} \) is shorthand for \( (\delta_1)^*; \bot \), for which Definition 5.16 implies that the set of nodes consist of nodes of the form \( \langle \bot i^{\omega}, \bot, \varphi_i' \rangle \) and \( (\bot ?, \bot) \).

It can further be argued that (5.37) is not a rigorous restriction because it represents the typical structure one would expect a robot’s non-terminating control program to have, where the intuition is that the robot concurrently performs a number of different (open-ended) tasks, each of which is implemented by some infinitely looping subprogram \( \delta_i^{\omega_i} \), and where one of the \( \delta_i^{\omega_i} \) moreover is the \( \delta_{\text{exo}} \) program that encodes exogenous actions. Obviously, our coffee robot’s program \( \delta_{\text{coffee}} \| \delta_{\text{exo}} \) is an example that adheres to this form.

\(^7\)We deal with such formulas in Section 5.4.
5.3.2 The Algorithm

We now define our meta-level operator that, similar to regression, transforms a formula of \( \mathcal{ESG}_{\text{CTL}} \) into an equivalent fluent formula. As all such manipulations will be with respect to some basic action theory \( \Sigma \), we first have the following definition:

**Definition 5.25** (Verifiable \( \mathcal{ESG}_{\text{CTL}} \) Formulas). Let \( \Sigma \) be a basic action theory over \( \langle D, F \rangle \).

An \( \mathcal{ESG}_{\text{CTL}} \) formula \( \varphi \) is verifiable iff it only mentions fluents from \( \langle D, F \rangle \), every quantifier uses a distinct variable and every occurrence of a functional fluent is of the form \( f(\vec{t}) = \vec{t}' \), where \( \vec{t} \) and \( \vec{t}' \) do not contain any further functional fluents.

Note that the restriction to fluents from \( \langle D, F \rangle \) also applies to test formulas within programs mentioned in \( \varphi \).

**Definition 5.26** (Verification Transform Operator for \( \mathcal{ESG}_{\text{CTL}} \) Formulas). Let \( \Sigma \) be a basic action theory over \( \langle D, F \rangle \) and \( \varphi \) a verifiable \( \mathcal{ESG}_{\text{CTL}} \) formula. The verification transformation of \( \varphi \) wrt \( \Sigma \), denoted as \( C[\varphi] \), is inductively defined as follows:

1. \( C[(t_1 = t_2)] = R[(t_1 = t_2)] \);
2. \( C[F(\vec{t})] = R[F(\vec{t})] \);
3. \( C[\varphi_1 \land \varphi_2] = C[\varphi_1] \land C[\varphi_2] \);
4. \( C[\neg \varphi] = \neg C[\varphi] \);
5. \( C[\exists x.\varphi] = \exists x.C[\varphi] \);
6. \( C[\langle\langle \delta \rangle\rangle X \varphi] = \text{CheckEX}[\delta, C[\varphi]] \);
7. \( C[\langle\langle \delta \rangle\rangle G \varphi] = \text{CheckEG}[\delta, C[\varphi]] \);
8. \( C[\langle\langle \delta \rangle\rangle (\varphi_1 U \varphi_2)] = \text{CheckEU}[\delta, C[\varphi_1], C[\varphi_2]] \).

The interesting cases of course are those for \( \langle\langle \delta \rangle\rangle X \varphi \), \( \langle\langle \delta \rangle\rangle G \varphi \) and \( \langle\langle \delta \rangle\rangle (\varphi_1 U \varphi_2) \) subformulas. For each of them, there is a corresponding procedure (\( \text{CheckEX} \), \( \text{CheckEG} \), and \( \text{CheckEU} \) respectively) which we will discuss in the following. First of all, each of these procedures works on the characteristic graph of the input program, using labels on the graph nodes:

**Definition 5.27** (Labels). Let \( G = \langle V, E, v_0 \rangle \) be a trace graph. A label on \( G \) is of the form \( \langle v, \psi \rangle \), where \( v \in V \) and \( \psi \) is a fluent formula. Furthermore, a labelling of \( G \) or a label set on \( G \) is a set of labels, containing exactly one label for each \( v \in V \).

---

8A labelling thus is actually a mapping from nodes to formulas, but for convenience, we will resort to a set-based notation.
Intuitively, a label represents a set of graph configurations. Formally, let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory. Then we define

$$\langle v, \psi \rangle_{\Sigma} \overset{\text{def}}{=} \{ \langle v, \theta, w, z \rangle \mid w, z \models \psi \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \}$$  \hspace{1cm} (5.41)$$

For any label set $L$ let

$$\| L \|_{\Sigma} \overset{\text{def}}{=} \bigcup \{ \| \langle v, \psi \rangle \|_{\Sigma} \mid \langle v, \psi \rangle \in L \}$$  \hspace{1cm} (5.42)$$

Definition 5.28 (Operations on Labels). We define the following operations on labels: Let $G = \langle V, E, v_0 \rangle$ be a trace graph, $\alpha$ a fluent formula and $L_1$ and $L_2$ label sets. Then

$$\text{Label}[\langle V, E, v_0 \rangle, \alpha] \overset{\text{def}}{=} \{ \langle v, \alpha \rangle \mid v \in V \}$$  \hspace{1cm} (5.43)$$

$$L_1 \text{ AND } L_2 \overset{\text{def}}{=} \{ \langle v, \psi_1 \land \psi_2 \rangle \mid \langle v, \psi_1 \rangle \in L_1, \langle v, \psi_2 \rangle \in L_2 \}$$  \hspace{1cm} (5.44)$$

$$L_1 \text{ OR } L_2 \overset{\text{def}}{=} \{ \langle v, \psi_1 \lor \psi_2 \rangle \mid \langle v, \psi_1 \rangle \in L_1, \langle v, \psi_2 \rangle \in L_2 \}$$  \hspace{1cm} (5.45)$$

$$\text{InitLabel}[\langle V, E, v_0 \rangle, L] \overset{\text{def}}{=} \psi \text{ such that } \langle v_0, \psi \rangle \in L$$  \hspace{1cm} (5.46)$$

That is we can generate a labelling for some specified formula $\alpha$ using $\text{Label}[G, \alpha]$, yielding a label set where every node in $G$ is labelled with $\alpha$. Conjunction and disjunction of two label sets is defined as expected, i.e. by applying these operations individually to the formulas in labels at the same node. $\text{InitLabel}[G, L]$ extracts the formula by which the initial node of $G$ is labelled (recall that there is exactly one label for each node, hence the returned formulas is unique). We furthermore need to decide when two label sets are equivalent:

Definition 5.29 (Label Equivalence). Let $L_1$ and $L_2$ be label sets. Then we define

$$L_1 \equiv L_2 \text{ iff for all } v \in V \text{ such that } \langle v, \psi_1 \rangle \in L_1 \text{ and } \langle v, \psi_2 \rangle \in L_2, \models \psi_1 \equiv \psi_2.$$  \hspace{1cm} (5.47)$$

We note that the conjunction, disjunction and equivalence of label sets corresponds to the intersection, union and equality, respectively, of the represented sets of configurations:

Lemma 5.30. Let $L_1$ and $L_2$ be label sets over some trace graph $G$ and $\Sigma$ a basic action theory. Then

$$\| L_1 \text{ AND } L_2 \|_{\Sigma} = \| L_1 \|_{\Sigma} \cap \| L_2 \|_{\Sigma}$$

$$\| L_1 \text{ OR } L_2 \|_{\Sigma} = \| L_1 \|_{\Sigma} \cup \| L_2 \|_{\Sigma}$$

$$L_1 \equiv L_2 \text{ iff } \| L_1 \|_{\Sigma} = \| L_2 \|_{\Sigma}$$
Proof. The first two cases follow immediately from (5.42) and Definition 5.28. In the third case, $|L_1|_\Sigma$ equals $\|L_2\|_\Sigma$ for any BAT $\Sigma$ due to the fact that labels contain only fluent formulas, which means that their truth (and hence by Definition 5.29 also equivalence between them) does not depend on $\Sigma_{\text{def}} \cup \Sigma_{\text{post}}$.

Finally, a paramount ingredient for all three subprocedures is the computation of the preimage of a set of labels:

Definition 5.31 (Preimage). Let $L$ be a set of labels on some trace graph $G = \langle V, E, v_0 \rangle$ and $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory. The preimage of $L$ is defined as:

\begin{equation}
\text{Pre}[\langle V, E, v_0 \rangle, L] = \{ (v, \text{Pre}[v, L]) \mid v \in V \}
\end{equation}

where

\begin{equation}
\text{Pre}[v, L] = \bigvee \{ R[\phi \land \exists \bar{x}. \phi' \land [t]\psi] \mid v \xrightarrow{\phi/\pi \bar{x} \lor \phi'} v' \in E, \langle v', \psi \rangle \in L \}.
\end{equation}

We follow the usual convention that the disjunction over the empty set is identified with $\bot$.

Here we consider all incoming edges of any node in the graph, where in each case, we have to determine the regression of the label formula of the adjacent node through the corresponding action. Furthermore included are the two transition conditions $\phi$ and $\phi'$ as well as an existential quantification over the $\pi$-quantified variables at the edge. Recall that $\phi$ is the condition that has to hold before instantiating the $\bar{x}$, hence the scope of $\exists \bar{x}$ begins only after $\phi$. Also note that it is necessary to regress $\phi$ and $\phi'$ because they may contain definitional fluents such as $\text{Poss}$.

Semantically, the preimage of $L$ represents the set of possible predecessor configurations of those represented by $L$, which is established in the following lemma:

Lemma 5.32. Let $G = \langle V, E, v_0 \rangle$ be a trace graph and $\Sigma$ be a basic action theory. Then

$|\text{Pre}[G, L]|_\Sigma = \{ (v, \theta, w, z) \mid (v, \theta, w, z) \xrightarrow{G} (v', \theta', w, z \cdot p), \langle v', \theta', w, z \cdot p \rangle \in |L|_\Sigma \}.$

Proof. \begin{align*}
(v, \theta, w, z) &\in |\text{Pre}[G, L]|_\Sigma \\
\text{iff} \quad (v, \theta, w, z) &\in |\langle v, \text{Pre}[v, L] \rangle|_\Sigma \quad \text{(by (5.48) and (5.42))} \\
\text{iff} \quad w, z &\models \text{Pre}[v, L] \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.41))} \\
\text{iff} \quad w, z &\models R[\phi \land \exists \bar{x}. \phi' \land [t]\psi] \theta, \\
&\text{where } v \xrightarrow{\phi/\pi \bar{x} \lor \phi'} v' \in E, \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.49))} \\
\text{iff} \quad w, z &\models \phi \theta \text{ and for some } \bar{u}, w, z \models \phi' \bar{u} \theta \text{ and } w, z \cdot p \models \psi \bar{u} \theta, \\
&\text{where } v \xrightarrow{\phi/\pi \bar{x} \lor \phi'} v' \in E, \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and } p = |t\bar{u}^\theta|_w \text{ (see below)}
\end{align*}
iff \( \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta[\vec{x}/\vec{n}], w, z \cdot p \rangle \),
\[ w, z \cdot p \models \psi\theta[\vec{x}/\vec{n}], \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \] (by Definition 5.15)
iff \( \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta[\vec{x}/\vec{n}], w, z \cdot p \rangle \) and \( \langle v', \theta[\vec{x}/\vec{n}], w, z \cdot p \rangle \in \|L\| : \Sigma \) (by (5.41))

Above, we used the following property about the regression of formulas and variable substitutions: If \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), then
\[ w, z \models \mathcal{R}[\alpha] \theta \iff w, z \models \alpha \theta, \]
which can be easily proven by an induction on \( \alpha \), similar to Lemma 3.32 item 4. Intuitively, this is correct since regression leaves the free variables of a formula unchanged. We also used the fact that since the \( \vec{x} \) are bound in \( \exists \vec{x} \alpha \),
\[ w, z \models (\exists \vec{x} \alpha) \theta \iff \text{for some } \vec{n}, w, z \models \alpha_{\vec{n}} \theta, \]
which follows directly from the \( \mathcal{ES}\mathcal{G} \) semantics and our definition of variable substitutions.

The “Next” Operator

We begin with the seemingly easiest case, which is the procedure to evaluate path-quantified formulas involving the “next” operator \( X \). The algorithm for \textsc{CheckEX} works as follows: If \( \langle \delta \rangle X \varphi \) is the input formula, then we determine the characteristic graph of the program \( \delta \) and initially label each of its vertices with \( \varphi \), conjoined with a formula that encodes all necessary and sufficient conditions under which an infinite trace may start in a configuration that corresponds to that node. Next, we determine the preimage of that label set, and finally extract the formula in the label of the initial node. Formally:

\[ \text{Procedure 1 } \textsc{CheckEX}[\delta, \varphi] \]
\begin{enumerate}
  \item \( L' := \text{Label}[G_\delta, \varphi] \) AND \text{Path}[G_\delta];
  \item \( L := \text{Pre}[G_\delta, L'] \);
  \item return \text{InitLabel}[G_\delta, L]
\end{enumerate}

Regarding \text{Path}[G_\delta], for the time being we simply assume that it returns a label set that encodes the necessary and sufficient conditions for any node \( v \) under which there exists an infinite run starting in \( v \). Formally let us assume that
\[ \langle v, \theta, w, z \rangle \in \text{Path}[G_\delta] \cap \Sigma \iff w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run} \]
\[ \langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G_\delta} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G_\delta} \ldots \]
It will be shown later how this label set can actually be computed.

**Example 5.33.** Let us apply the algorithm to the input formula $\langle\langle \delta \rangle\rangle X Empty(queue)$, where $\delta$ refers to $(\delta_{coffee} \cdot \delta_{exo})^p$, i.e. the precondition-extended variant of $\delta_{coffee} \cdot \delta_{exo}$. Recall from Definition 5.11 that this means that we replace every atomic action $t$ by $Poss(t)?t$. The resulting characteristic graph, which is depicted in Figure 5.3, is nearly identical to the one shown in Figure 5.2, with the exception that the transition condition of every edge also contains a test for $Poss(t)$. According to Definition 5.26, we get that

$$C[\langle\langle \delta \rangle\rangle X Empty(queue)] = \text{CHECKEX}[\delta, C[Empty(queue)]] = \text{CHECKEX}[\delta, Empty(queue)].$$

To evaluate the latter, presume that $\text{PATH}[G_\delta] = \{(v_0, \top), (v_1, \neg HoldingCoffee), (v_2, HoldingCoffee)\}$ and that $\text{CHECKEX}$ hence begins with the following label set:

$$L' = \{(v_0, Empty(queue)), (v_1, Empty(queue) \land \neg HoldingCoffee), (v_2, Empty(queue) \land HoldingCoffee)\}.$$

Next, the preimage of this set is determined. Below we only show the computation of the resulting label for $v_0$ in detail, since the other ones are not relevant for the end result. The preimage for $v_0$, $\text{PRE}[G_\delta, v_0]$ consists of three disjuncts, one for each outgoing edge:

- For the $v_0 \xrightarrow{\text{Exo}/a \cdot \text{Poss}(a)} v_0$ edge, we get (with simplifications):
  $$\mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]Empty(queue)]$$
  $$\equiv \exists a. a = \text{requestCoffee}(x) \land x \neq e \land \text{LastFree}(queue) \land \mathcal{R}[[a]Empty(queue)]$$
  $$\equiv \exists x. x \neq e \land \text{LastFree}(queue) \land \mathcal{R}[[\text{requestCoffee}(x)]Empty(queue)]$$
  $$\equiv \exists x. x \neq e \land \text{LastFree}(queue) \land \text{Enqueue}(queue, x, \langle e, \ldots, e \rangle)$$
  $$\equiv \exists x. x \neq e \land \text{LastFree}(queue) \land \bot$$
  $$\equiv \bot$$

- For the $v_0 \xrightarrow{\text{wait}/\cdots} v_0$ edge we have (with simplifications):
  $$\mathcal{R}[Empty(queue) \land \text{Poss}(wait) \land [\text{wait}]Empty(queue)]$$
  $$\equiv Empty(queue) \land \top \land \mathcal{R}[[\text{wait}]Empty(queue)]$$
  $$\equiv Empty(queue) \land \top \land Empty(queue)$$
  $$\equiv Empty(queue)$$
5.3 Nonterminating Programs

For the \( v_0 \xrightarrow{\pi x : \text{selectRequest}(x)/\neg\text{Empty}(\text{queue}) \land \text{Poss}(sR(x))} v_1 \) edge we get (with simplifications):

\[
R[\exists x. \neg\text{Empty}(\text{queue}) \land \text{Poss}(sR(x))] \\
\equiv \exists x. (\neg\text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(x, \text{queue}) \land R[[sR(x)](\text{Empty}(\text{queue}) \land \neg\text{HC})]) \\
\equiv \exists x. (\neg\text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(x, \text{queue}) \land \text{Dequeue}(\text{queue}, x, \langle e, \ldots, e \rangle) \land \neg\text{HC}) \\
\equiv \exists x. (\neg\text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(x, \text{queue}) \land \text{queue} = \langle x, e, \ldots, e \rangle \land \neg\text{HC}) \\
\equiv \exists x. x \neq e \land \text{queue} = \langle x, e, \ldots, e \rangle \land \neg\text{HC}
\]

The resulting label set is hence:

\[
L = \{ (v_0, \text{Empty}(\text{queue}) \lor \exists x. x \neq e \land \text{queue} = \langle x, e, \ldots, e \rangle \land \neg\text{HC}), (v_1, \ldots), (v_2, \ldots) \}
\]

As a result, the procedure returns the formula in the \( v_0 \) label, i.e.

(5.53) \( \text{Empty}(\text{queue}) \lor \exists x. x \neq e \land \text{queue} = \langle x, e, \ldots, e \rangle \land \neg\text{HoldingCoffee} \)

As will be proven formally later, this sentence represents precisely the necessary and sufficient conditions under which \( X\text{Empty}(\text{queue}) \) holds for some execution trace of the program \( \delta \). Intuitively, this is true since the two disjuncts represent exactly the two cases under which the queue can be empty after the first action and it is still possible to continue the execution of the program: either the action is \( \text{wait} \), which does not have any effect at all, and the queue was already empty in the beginning, or there initially was one request in the queue and the
action is to select that request to be served next, which removes it from the queue. In the first case, program execution may afterwards continue indefinitely with further wait actions that do not require any more preconditions to hold. In the latter case, we have to ensure that the robot is not already holding coffee such that we may subsequently proceed with pickuCoffee and bringCoffee(x), after which the program can again continue indefinitely with wait actions. Note that it is impossible that the queue is empty after an exogenous action since that would be a requestCoffee action appending some request to the queue.

**Theorem 5.34.** Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over \( \langle D, F \rangle \), \( \delta \) a program of the form \( \delta_1 \omega \cdots \delta_k \omega \) that only mentions fluents in \( \langle D, F \rangle \), and \( \varphi \) a fluent formula wrt \( \langle D, F \rangle \). If the procedure terminates, \( \text{CHECKEX}[\delta, \varphi] \) is fluent and

\[
\Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \text{CHECKEX}[\delta, \varphi] \equiv \langle \langle \delta \rangle \rangle X \varphi
\]

**Proof.** We prove the equivalent claim that for all \( w \in W \) with \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), all \( z \in Z \) and all variable maps \( \theta \),

\[
w, z \models \text{CHECKEX}[\delta, \varphi] \theta \text{ iff } w, z \models \langle \langle \delta \theta \rangle \rangle X \varphi \theta.
\]

To see why we need the variable map \( \theta \) here, note that \( \delta \) and \( \varphi \) as well as \( \text{CHECKEX}[\delta, \varphi] \) may contain free variables, and recall our convention that we regard free variables in formulas as being implicitly \( \forall \)-quantified from the outside. Now assume that Procedure 1 terminates.\(^9\)

Semantically, we have that

\[
|L'|_\Sigma = \{ \langle v, \theta, w, z \rangle \mid w, z \models \varphi \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \} \cap \| \text{PATH}[G_\delta] \|_\Sigma
\]

\[
\|L\|_\Sigma = \| \text{PRE}[G_\delta, L'] \|_\Sigma
\]

We furthermore need that

\[
\text{if } \langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v', \theta', w, z \cdot p \rangle, \text{ then } \varphi \theta = \varphi \theta'.
\]

This is true since \( \theta' \) is of the form \( \theta[\overline{x}/\overline{n}] \), where \( v \xrightarrow{\phi_1/\pi \overline{x} \cdot t/\phi_2} v' \) is an edge in \( G_\delta \). Since the \( \overline{x} \) are \( \pi \)-quantified in \( \delta \), due to our distinctiveness assumption they do not appear freely in \( \varphi \), therefore applying \( \theta \) or \( \theta' \) yields the same result. Now let \( w \) be a world with \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). Then

\[
w, z \models \text{CHECKEX}[\delta, \varphi] \theta
\]

iff \( w, z \models \text{INITLABEL}[G_\delta, L] \theta \) (by assumption)

iff \( \langle \psi_0, \psi \rangle \in \text{PRE}[G_\delta, L'], w, z \models \psi \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) (by (5.55) and (5.46))

\(^9\)As we will see shortly, non-termination may be due to the computation of \( \text{PATH}[G_\delta] \) not converging.
iff \((v_0, \theta, w, z) \in \|\text{Pre}[G\delta, L']\|_\Sigma\) (by (5.41))
iff \((v_0, \theta, w, z) \xrightarrow{G\delta} (v_1, \theta_1, w, z \cdot p_1)\) and \((v_1, \theta_1, w, z \cdot p_1) \in [L']\|_\Sigma\) (by Lemma 5.32)
iff \((v_0, \theta, w, z) \xrightarrow{G\delta} (v_1, \theta_1, w, z \cdot p_1), w, z \cdot p_1 = \varphi_1\) and
\((v_1, \theta_1, w, z \cdot p_1) \xrightarrow{G\delta} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \xrightarrow{G\delta} \ldots\) (by (5.54) and (5.52))
iff \((v_0, \theta, w, z) \xrightarrow{G\delta} (v_1, \theta_1, w, z \cdot p_1), w, z \cdot p_1 = \varphi_1\) and
\((v_1, \theta_1, w, z \cdot p_1) \xrightarrow{G\delta} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \xrightarrow{G\delta} \ldots\) such that
\(w, z \cdot p_1 = \varphi\) and for all \(i \geq 0\), \((v_i, \theta_i, w, z \cdot p_1 \cdots p_i) \notin F_{G\delta}\) (by Lemma 5.24)
iff \(\pi \in \|G\delta\|_\omega(\theta)\) and \(w, z, \pi = \varphi\) (by Definition 5.15 and the semantics)
iff \(\pi \in \|\delta\|_\omega\) and \(w, z, \pi = \varphi\) (by Theorem 5.21)
iff \(w, z = \langle\langle \delta\rangle\rangle \varphi\) (by the semantics)

Finally, since labels always contain fluent formulas, the output of the procedure (if any) is also a fluent formula.

**The “Always” Operator**

Let us turn to the more interesting case of the “always” operator \(G\). The idea behind the corresponding procedure is as follows. To verify \(\langle\langle \delta\rangle\rangle \varphi\), we first label each vertex in the characteristic graph of \(\delta\) with \(\varphi\), which represents all configurations where \(\varphi\) holds. Next, we determine the preimage of this labelling and conjoin it with the initial one, yielding a representation of the set of all configurations where \(\varphi\) holds and will hold in the successor configuration. Iterating this, we successively obtain representations of configurations from which on \(\varphi\) persists to hold for 2, 3, … steps of a partial run. Convergence is reached once the newly determined labelling is equivalent to the previous one. The initial node’s label then contains the necessary and sufficient condition under which an infinite run exists along which \(\varphi\) never ceases to be true. Formally:

**Procedure 2** CHECKEG[, \(\varphi\)]

1. \(L' := \text{LABEL}[G\delta, \bot]\);
2. \(L := \text{LABEL}[G\delta, \varphi]\);
3. while \(L \neq L'\) do
   4. \(L' := L\);
5. \(L := L'\) And \(\text{Pre}[G\delta, L']\);
6. end while
7. return \(\text{INITLABEL}[G\delta, L]\)
Example 5.35. Let us apply the procedure within our coffee robot example to verify one of the two previously mentioned properties, namely whether it is possible that no request is ever served. Recall that this is expressed by the formula

\[
\langle\langle \delta \rangle\rangle_G \neg \exists x \text{Occ}(\text{selectRequest}(x))
\]

That is, let \(\delta\) again be the precondition-extended variant of \(\delta_{\text{coffee}}\left[\delta_{\text{exo}}\right]\) and \(\varphi\) the formula \(\neg \exists x \text{Occ}(\text{selectRequest}(x))\). Furthermore, in order to keep the example small, assume that the size of the robot’s queue is \(k = 2\). Then the initial set of labels is

\[
L_0 = \{(v_0, \varphi), (v_1, \varphi), (v_2, \varphi)\}
\]

which is obviously not equivalent to \(\text{Label}[[G_\delta, \bot]]\). We need to determine the preimage of each node. Note that due to the unique names assumption for actions, \(\mathcal{R}[[a]\varphi]\) is equivalent to \(\bot\) just in case the action \(a\) is an instance of \(\text{selectRequest}(x)\), and otherwise \(\top\). That is \(\mathcal{R}[[\text{requestCoffee}(x')]\varphi] \equiv \top\), \(\mathcal{R}[[\text{selectRequest}(x)]]\varphi \equiv \bot\), and \(\mathcal{R}[[\text{wait}]]\varphi \equiv \top\). We thus get (with simplifications):

- **Pre\([v_0, L_0]\)**
  \[
  \equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]\varphi] \lor
  \mathcal{R}[[\exists x. \neg \text{Empty}(\text{queue}) \land \text{Poss}(\text{selectRequest}(x)) \land [\text{selectRequest}(x)]\varphi] \lor
  \mathcal{R}[[\text{Empty}(\text{queue}) \land \text{Poss}(\text{wait})] \land [\text{wait}]\varphi]
  \]
  \[
  \equiv \exists a \exists x'(a = \text{requestCoffee}(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top) \lor
  \exists x(\neg \text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(\text{queue}, x) \land \bot) \lor
  \text{Empty}(\text{queue}) \land \top \land \top
  \]
  \[
  \equiv \text{LastFree}(\text{queue}) \lor \bot \lor \text{Empty}(\text{queue})
  \]
  \[
  \equiv \exists x_1(\text{queue} = \langle x_1, e \rangle) \lor \text{queue} = \langle e, e \rangle
  \]
  \[
  \equiv \exists x_1(\text{queue} = \langle x_1, e \rangle)
  \]
  \[
  \equiv \text{LastFree}(\text{queue})
  \]

- **Pre\([v_1, L_0]\)**
  \[
  \equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]\varphi] \lor \mathcal{R}[[\text{Poss}(\text{pickupCoffee}) \land [\text{pickupCoffee}]\varphi]
  \]
  \[
  \equiv \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top) \lor \neg \text{HoldingCoffee} \land \top
  \]
  \[
  \equiv \text{LastFree}(\text{queue}) \lor \neg \text{HoldingCoffee}
  \]
5.3 Nonterminating Programs

\[ \text{Pre}[v_2, L_0] \equiv \mathcal{R} [\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a] \varphi] \lor \mathcal{R} [\text{Poss} \text{bringCoffee}(x) \land [\text{bringCoffee}(x)] \varphi] \]

\[ \equiv \exists a \exists x' (a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top) \lor \text{HoldingCoffee} \land \top \]

\[ \equiv \text{LastFree}(\text{queue}) \lor \text{HoldingCoffee} \]

In the simplification steps above, we further used that if a formula starts with

\[ \exists a \exists x' (a = \text{requestCoffee}(x') \land x' \neq e \cdots) , \]

then that part is trivially satisfied if \( a \) and \( x' \) do not appear in the remainder of the formula, and that \( \text{Empty}(\text{queue}) \) implies \( \text{LastFree}(\text{queue}) \), which follows from the definition of these abbreviations as stated in (5.19) and (5.21). After one iteration, our (simplified) label set is hence

\[ L_1 = L_0 \text{ AND Pre}[G_δ, L_0] \]

\[ = \{ \langle v_0, \varphi \land \text{LastFree}(\text{queue}) \rangle, \langle v_1, \varphi \land (\text{LastFree}(\text{queue}) \lor \neg \text{HC}) \rangle, \langle v_2, \varphi \land (\text{LastFree}(\text{queue}) \lor \text{HC}) \rangle \} \]

Obviously, \( L_1 \) is not equivalent to \( L_0 \), and we have to continue. Doing four more iterations (the details can be found in Appendix B.1), we end up with the label set

\[ L_5 = \{ \langle v_0, \varphi \land \text{Empty}(\text{queue}) \rangle, \langle v_1, \varphi \land \text{Empty}(\text{queue}) \land \neg \text{HC} \rangle, \langle v_2, \varphi \land \text{Empty}(\text{queue}) \land \text{HC} \rangle \} \]

One last iteration will show us that convergence is reached, i.e. we have \( L_6 \equiv L_5 \). The algorithm hence terminates and outputs the label at \( v_0 \), which is the formula

\[ \neg \exists x \text{Occ}(\text{selectRequest}(x)) \land \text{Empty}(\text{queue}) \]

Intuitively, this correctly represents the necessary and sufficient conditions under which there is an execution of \( δ \) where never any \( \text{selectRequest}(x) \) action occurs: On the one hand, it has to be the case that there did not occur a \( \text{selectRequest}(x) \) just before the current situation \( z \) in which we are about to start the program, as then \( \text{Occ}(\text{selectRequest}(x)) \) would hold in \( z \). On the other hand, the queue of pending coffee requests must be empty, since only then we may execute \( \text{wait} \) actions indefinitely, provided that no exogenous \( \text{requestCoffee}(x) \) ever occurs.

**Theorem 5.36.** Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over \( (D, F) \), \( δ \) a program of the form \( δ_1^ω | \cdots | δ_k^ω \) that only mentions fluents in \( (D, F) \), and \( \varphi \) a fluent formula wrt \( (D, F) \). If the procedure terminates, \( \text{CheckEG}[δ, \varphi] \) is fluent and

\[ \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \text{CheckEG}[δ, \varphi] \equiv \langle \langle δ \rangle \rangle \mathcal{G} \varphi \]
Proof. We will prove the equivalent claim that for all \( w \in W \) such that \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), all \( z \in Z \) and all variable maps \( \theta \),

\[
w, z \models \text{CHECKEG}[\delta, \varphi] \text{ iff } w, z \models \langle \langle \delta \theta \rangle \rangle G \varphi \theta.
\]

Again note that \( \theta \) is needed since \( \delta \) and \( \varphi \), and consequently \( \text{CHECKEG}[\delta, \varphi] \), may contain free variables, which by convention we regard as being implicitly \( \forall \)-quantified from the outside.

First consider the simple case where \( \varphi \) is unsatisfiable. Then \( \models \varphi \equiv \bot \), and hence \( L \equiv L' \), so the procedure does not enter the while loop, but immediately returns \( \bot \) as output. The claim then trivially follows since neither \( w, z \models \bot \) nor can there be any trace \( \tau \) such that \( w, z, \tau \models G \bot \).

Therefore assume now that \( \varphi \) is satisfiable and that there is at least one iteration of the while loop. In this case the algorithm iteratively constructs label sets according to the following equations:

\[
\begin{align*}
L_0 &= \text{LABEL}[G_\delta, \varphi] \\
L_{i+1} &= L_i \text{ AND PRE}[G_\delta, L_i]
\end{align*}
\]

Semantically, this means that

\[
\begin{align*}
|L_0|_\Sigma &= \{ \langle v, \theta, w, z \rangle \mid w, z \models \varphi \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \} \\
|L_{i+1}|_\Sigma &= |L_i|_\Sigma \cap |\text{PRE}[G_\delta, L_i]|_\Sigma
\end{align*}
\]

By assumption the procedure terminates, therefore there is some smallest index \( n \) such that \( L_{n+1} \equiv L_n \). Together with the above this implies that

\[
|L_n|_\Sigma = |L_n|_\Sigma \cap |\text{PRE}[G_\delta, L_n]|_\Sigma.
\]

The output of the procedure is obviously a fluent formula: We start off with all labels containing \( \varphi \), which is fluent by assumption, and subsequently apply conjunction and regression, which always leads to label formulas that are fluent as well.

The proof of the main claim of this theorem is based on the following property, which will be proved below:

\[
\langle v, \theta, w, z \rangle \in |L_n|_\Sigma \text{ iff } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run}
\]

\[
\langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G_\delta} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G_\delta} \ldots
\]

such that for all \( i \geq 0 \), \( w, z \cdot p_1 \cdots p_i \models \varphi \theta \)

Let \( w \) be a world with \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). Then we get that
\[ w, z \models \text{CHECKEG}[\delta, \varphi] \theta \]

\begin{align*}
&\text{iff } w, z \models \text{INITLABEL}[G_\delta, L_{n+1}] \theta \\
&\text{iff } (v_0, \psi_0) \in L_{n+1} \text{ and } w, z \models \psi_0 \theta \\
&\text{iff } (v_0, \theta, w, z) \in \|L_{n+1}\| \Sigma \\
&\text{iff } (v_0, \theta, w, z) \in \|L_n\| \Sigma \\
&\text{iff } (v_0, \theta, w, z) \overset{G_\delta}{\rightarrow} (v_1, \theta_1, w, z \cdot p_1) \overset{G_\delta}{\rightarrow} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \overset{G_\delta}{\rightarrow} \ldots \\
&\text{such that for all } i \geq 0, (v_i, \theta_i, w, z \cdot p_1 \ldots p_i) \notin \mathcal{F}^{G_\delta}, \text{ and } w, z \cdot p_1 \cdots p_i \models \varphi \theta
\end{align*}

(by (5.62) and Lemma 5.24)

\begin{align*}
&\text{iff } \pi \in \|[G_\delta]_w^z(\theta)\| \text{ with } w, z, \pi \models G \varphi \theta \\
&\text{iff } \pi \in \|\delta \|_w^z \text{ with } w, z, \pi \models G \varphi \theta \\
&\text{iff } w, z \models \{\langle \delta \rangle \} G \varphi \theta
\end{align*}

(by Definition 5.15 and the semantics)

(by Theorem 5.21)

(by the semantics)

To prove (5.62), first note that

\[ (5.63) \quad \text{if } (v, \theta, w, z) \overset{G_\delta}{\rightarrow} (v', \theta', w, z \cdot p), \text{ then } \varphi \theta = \varphi \theta'. \]

As in the proof to Theorem 5.34, this is due to the fact that \( v \overset{\phi_1/\pi \bar{x}/\bar{n}}{\rightarrow} v' \) is an edge in \( G_\delta \) such that \( \theta' = \theta[\bar{x}/\bar{n}] \). Since variables that appear \( \pi \)-quantified in \( \delta \) are assumed to not appear freely in \( \varphi \), applying \( \theta \) or \( \theta' \) yields the same result.

\[ \Rightarrow \]: For the only-if direction, suppose that \( (v, \theta, w, z) \in \|L_n\| \Sigma \). An infinite run with the desired properties is guaranteed to exist using the following: For any \( (v', \theta', w', z') \in \|L_n\| \Sigma \),

\begin{enumerate}
  \item \( w', z' \models \varphi \theta' \) and \( w' \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \);
  \item \( (v', \theta', w', z') \overset{G_\delta}{\rightarrow} (v'', \theta'', w', z' \cdot p) \) such that \( (v'', \theta'', w', z' \cdot p) \in \|L_n\| \Sigma \).
\end{enumerate}

(a) is obtained by repeated application of (5.60), which implies that any configuration in \( \|L_{i+1}\| \Sigma \) is also in \( \|L_i\| \Sigma \), hence any \( (v', \theta', w', z') \) in \( \|L_n\| \Sigma \) is also in \( \|L_0\| \Sigma \), and therefore by (5.59), \( w', z' \models \varphi \theta' \) and \( w' \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). Furthermore, (b) is a direct consequence of (5.61) and Lemma 5.32. With the above, we inductively construct an infinite run whose configurations remain within \( \|L_n\| \Sigma \): By assumption, \( (v, \theta, w, z) \) is in \( \|L_n\| \Sigma \), and for every such configuration, (b) implies that there is a transition to a successor configuration that is in \( \|L_n\| \Sigma \) as well. Furthermore, in any situation thus reached, \( \varphi \theta' \) is satisfied according to (a), which is the same as \( \varphi \theta \) according to (5.63).

\[ \Leftarrow \]: Conversely, assume that \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) and that there is an infinite run

\[ (v, \theta, w, z) \overset{G_\delta}{\rightarrow} (v_1, \theta_1, w, z \cdot p_1) \overset{G_\delta}{\rightarrow} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \overset{G_\delta}{\rightarrow} \ldots \]

such that for all \( j \geq 0, w, z \cdot p_1 \cdots p_j \models \varphi \theta \). We use the following two properties:
(a) For all \( j \geq 0 \), \( \langle v_j, \theta_j, w, z \cdot p_1 \cdots p_j \rangle \in |L_0|_{\Sigma} \)

(b) If \( \langle v', \theta', w', z' \cdot p \rangle \in |L_i|_{\Sigma} \) and \( \langle v, \theta, w', z' \rangle \xrightarrow{\delta_i} \langle v', \theta', w', z' \cdot p \rangle \), then \( \langle v, \theta, w', z' \rangle \in |L_{i+1}|_{\Sigma} \).

(a) is a direct consequence of (5.59) and (5.63), whereas (b) follows from (5.60) and Lemma 5.32.

From the above we obtain that, in particular, the \( n \)-th graph configuration \( \langle v_n, \theta_n, w, z \cdot p_1 \cdots p_n \rangle \) is in \( |L_0|_{\Sigma} \), hence the \( n-1 \)-st configuration \( \langle v_{n-1}, \theta_{n-1}, w, z \cdot p_1 \cdots p_{n-1} \rangle \) is in \( |L_1|_{\Sigma} \), therefore \( \langle v_{n-2}, \theta_{n-2}, w, z \cdot p_1 \cdots p_{n-2} \rangle \in |L_2|_{\Sigma} \), and so forth, until we end up with \( \langle v, \theta, w, z \rangle \in |L_n|_{\Sigma} \), which was to be proved.

Before moving on to the “until” case, note that we can now easily define a procedure for the previously mentioned \( \text{Path}[G_\delta] \) function, which is quite similar to \( \text{CheckEG} \):

**Procedure 3 \( \text{Path}[G_\delta] \)**

1. \( L' := \text{Label}[G_\delta, \bot] \)
2. \( L := \text{Label}[G_\delta, \top] \)
3. while \( L \not\equiv L' \) do
   4. \( L' := L \)
   5. \( L := L' \) and \( \text{Pre}[G_\delta, L'] \)
4. end while
5. return \( L \)

There are only two differences here: On the one hand, we return the entire resulting label set, instead of only the formula by which \( v_0 \) is labelled. For another, the \( \varphi \) parameter has been replaced by \( \top \). Intuitively, this yields the desired result since \( \text{CheckEG} \) determines the necessary and sufficient conditions under which there is an infinite run through \( G_\delta \) starting from \( v_0 \) such that \( \varphi \) holds globally along the traversed path, whereas \( \text{Path}[G_\delta] \) is supposed to compute the necessary and sufficient conditions for each node \( v \) in \( G_\delta \) under which there exists an arbitrary infinite run starting in \( v \).

**Lemma 5.37.** The output of Procedure 3 satisfies property (5.52). That is, let \( \Sigma \) be a basic action theory and \( \delta \) a program of the form \( \delta_1 \| \cdots \| \delta_k \| \) without free variables. If the procedure terminates, \( \text{Path}[G_\delta] \) returns a label set \( L \) such that

\[
\langle v, \theta, w, z \rangle \in |L|_{\Sigma} \text{ iff } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run } \langle v, \theta, w, z \rangle \xrightarrow{\delta_1} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{\delta_2} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{\delta_3} \ldots
\]

**Proof.** The proof is identical to the one for property (5.62) in the proof of Theorem 5.36, only with \( \varphi \) replaced by \( \top \).
The “Until” Operator

In the case of a formula \((\varphi_1 U \varphi_2)\) containing the “until” operator, our algorithm proceeds as follows. First, label each vertex with \(\varphi_2\) together with the path existence condition obtained from \(\text{PATH}[G_\delta]\). Intuitively, this labelling represents all configurations where \(\varphi_2\) holds and from where we may continue an infinite run through the graph. We then determine the disjunction of that label set with the conjunction of its preimage and \(\varphi_1\), yielding a representation of the configurations where either \(\varphi_2\) holds (and then some infinite run starts), or where \(\varphi_1\) holds and a configuration satisfying \(\varphi_2\) is reachable in one step (after which an infinite run starts). When iterating this a second time, we obtain a set of labels that tell us under which conditions we can reach such a configuration in at most two steps, and so on. If the procedure eventually converges, we have found a solution that represents all necessary and sufficient conditions under which there exists and infinite run where \(\varphi_2\) holds after \text{any} number of transitions, and \(\varphi_1\) is true until then. Formally:

\[
\begin{align*}
\text{Procedure 4 } & \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \\
1: & \quad L' := \text{LABEL}[G_\delta, \top]; \\
2: & \quad L := \text{LABEL}[G_\delta, \varphi_2] \text{ AND } \text{PATH}[G_\delta]; \\
3: & \quad \text{while } L \neq L' \text{ do} \\
4: & \quad \quad L' := L; \\
5: & \quad \quad L := L' \text{ OR } (\text{LABEL}[G_\delta, \varphi_1] \text{ AND } \text{PRE}[G_\delta, L']); \\
6: & \quad \text{end while} \\
7: & \quad \text{return } \text{INITLABEL}[G, L]
\end{align*}
\]

Example 5.38. Again, let us apply the algorithm to an example. Suppose we want to verify the formula

\[
\langle\langle \delta \rangle\rangle (\text{Empty(queue)} U \text{HoldingCoffee})
\]

where \(\delta\) is, as before, the precondition-extended variant of our coffee robot control program, including exogenous actions, whose characteristic graph is depicted in Figure 5.3. Again we assume that the maximal size of the queue is \(k = 2\) and that

\[
\text{PATH}[G_\delta] = \{(v_0, \top), (v_1, \neg\text{HoldingCoffee}), (v_2, \text{HoldingCoffee})\}
\]

Then the initial set of labels is therefore

\[
\begin{align*}
L_0 &= \{(v_0, HC \land \top), (v_1, HC \land \neg HC), (v_2, HC \land HC)\} \\
&\equiv \{(v_0, HC), (v_1, \bot), (v_2, HC)\}
\end{align*}
\]
Since this is obviously not equivalent to the set
\[ L' = \{ \langle v_0, \top \rangle, \langle v_1, \top \rangle, \langle v_2, \top \rangle \}, \]
we have to first determine the preimage of this set:

- **Pre**[\(v_0, L_0\)]
  \[
  \equiv R[\exists a. Exo(a) \land Poss(a) \land [a] HoldingCoffee] \lor \\
  R[\exists x. \neg Empty(queue) \land Poss(selectRequest(x)) \land [selectRequest(x)] \bot] \lor \\
  R[Empty(queue) \land Poss(wait) \land [wait] HoldingCoffee] \\
  \equiv \exists a \exists x' (a = requestCoffee(x') \land x' \neq e \land LastFree(queue) \land HoldingCoffee) \lor \\
  \exists x (\neg Empty(queue) \land x \neq e \land IsFirst(queue, x) \land \bot) \lor \\
  Empty(queue) \land \bot \land HoldingCoffee \\
  \equiv LastFree(queue) \land HoldingCoffee \lor \bot \lor Empty(queue) \land HoldingCoffee \\
  \equiv LastFree(queue) \land HoldingCoffee
  \]

- **Pre**[\(v_1, L_0\)]
  \[
  \equiv R[\exists a. Exo(a) \land Poss(a) \land [a] \bot] \lor \\
  R[Poss(pickupCoffee) \land [pickupCoffee] HoldingCoffee] \\
  \equiv \bot \lor \neg HoldingCoffee \land \top \\
  \equiv \neg HoldingCoffee
  \]

- **Pre**[\(v_2, L_0\)]
  \[
  \equiv R[\exists a. Exo(a) \land Poss(a) \land [a] HoldingCoffee] \lor \\
  R[Poss(bringCoffee(x)) \land [bringCoffee(x)] HoldingCoffee] \\
  \equiv LastFree(queue) \land HoldingCoffee \lor HoldingCoffee \land \bot \\
  \equiv LastFree(queue) \land HoldingCoffee
  \]

Using the above, the label set after the first iteration is
\[
L_1 = L_0 \text{ OR } (\text{LABEL}[G_0, \varphi_1] \text{ AND } \text{Pre}[G_0, L_0]) \\
= \{ \langle v_0, HC \lor Empty(queue) \land LastFree(queue) \land HC \rangle, \\
  \langle v_1, \bot \lor Empty(queue) \land \neg HC \rangle, \\
  \langle v_2, HC \lor Empty(queue) \land LastFree(queue) \rangle \} \\
\equiv \{ \langle v_0, HC \rangle, \langle v_1, Empty(queue) \land \neg HC \rangle, \langle v_2, HC \lor Empty(queue) \rangle \}\]
Since \(L_1\) is not equivalent to \(L_0\), we have to do another iteration:

- **Pre\([v_0, L_1]\)**
  \[
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]HC] \lor \\
  R[\exists x. \neg \text{Empty}(\text{queue}) \land \text{Poss}(sR(x)) \land [sR(x)](\text{Empty}(\text{queue}) \land \neg HC)] \lor \\
  R[\text{Empty}(\text{queue}) \land \text{Poss}(\text{wait}) \land [\text{wait}]HC] \\
  \equiv \exists a \exists x'(a = \text{requestCoffee}(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land HC) \lor \\
  \exists x(\neg \text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(\text{queue}, x) \land \\
  \text{Empty}(\text{queue}) \land \top \land \text{HC} \\
  \equiv \text{LastFree}(\text{queue}) \land HC \lor \exists x(x \neq e \land \text{queue} = \langle x, e \rangle \land \neg HC) \lor \text{Empty}(\text{queue}) \land HC) \\
  \equiv \text{LastFree}(\text{queue}) \land \text{HoldingCoffee} \lor \exists x(x \neq e \land \text{queue} = \langle x, e \rangle \land \neg \text{HoldingCoffee})
  \]

- **Pre\([v_1, L_1]\)**
  \[
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\text{Empty}(\text{queue}) \land \neg \text{HoldingCoffee})] \lor \\
  R[\text{Poss}(\text{pickupCoffee}) \land [\text{pickupCoffee}](\text{HoldingCoffee} \lor \text{Empty}(\text{queue}))] \\
  \equiv \text{LastFree}(\text{queue}) \land \bot \land \neg \text{HoldingCoffee} \lor \neg \text{HoldingCoffee} \land (\top \lor \text{Empty}(\text{queue})) \\
  \equiv \bot \lor \neg \text{HoldingCoffee} \\
  \equiv \neg \text{HoldingCoffee}
  \]

- **Pre\([v_2, L_1]\)**
  \[
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\text{HoldingCoffee} \lor \text{Empty}(\text{queue}))] \lor \\
  R[\text{Poss}(\text{bringCoffee}(x)) \land [\text{bringCoffee}(x)]\text{HoldingCoffee}] \\
  \equiv \text{LastFree}(\text{queue}) \land (\text{HoldingCoffee} \lor \bot) \lor \text{HoldingCoffee} \land \bot \\
  \equiv \text{LastFree} \land \text{HoldingCoffee}
  \]
The label set after the second iteration thus is

\[ L_2 = L_1 \text{ OR (LABEL}[G_\delta, \varphi_1] \text{ AND PRE}[G_\delta, L_1]) \]

\[ = \{ v_0, HC \lor \text{Empty} (\text{queue}) \land \]

\[(\text{LastFree} (\text{queue}) \land HC \lor \exists x (x \neq e \land \text{queue} = (x, e) \land \neg HC)), \]

\[ (v_1, \text{Empty} (\text{queue}) \land \neg HC \lor \text{Empty} (\text{queue}) \land \neg HC), \]

\[ (v_2, HC \lor \text{Empty} (\text{queue}) \lor \text{Empty} (\text{queue}) \land \text{LastFree} (\text{queue}) \land HC) \} \]

\[ \equiv \{ (v_0, HC \lor \text{Empty} (\text{queue}) \land HC \lor \text{Empty} (\text{queue}) \land \bot \land \neg HC), \]

\[ (v_1, \text{Empty} (\text{queue}) \land \neg HC), \]

\[ (v_2, HC \lor \text{Empty} (\text{queue}) \lor \text{Empty} (\text{queue}) \land HC) \} \]

\[ \equiv \{ (v_0, HC), (v_1, \text{Empty} (\text{queue}) \land \neg HC), (v_2, HC \lor \text{Empty} (\text{queue})) \} \]

We now have \( L_2 \equiv L_1 \), so we are done. The output of the procedure is the label at \( v_0 \), i.e. the formula \( \text{HoldingCoffee} \). This is intuitively correct since the only possible way to execute the coffee robot program and maintain an empty queue of requests until picking up coffee is when the robot is already holding coffee initially. When then no requests ever occur, the robot can keep doing \text{wait} actions indefinitely. Otherwise, the queue will obviously not be empty once some \text{requestCoffee}(x) occurs.

**Theorem 5.39.** Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over \(\langle D, F \rangle \), \( \delta \) a program of the form \( \delta_1^\omega \| \cdots \| \delta_k^\omega \) that only mentions fluents in \(\langle D, F \rangle \), and \( \varphi_1, \varphi_2 \) fluent formulas wrt \(\langle D, F \rangle \). If the procedure terminates, \( \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \) is fluent and

\[ \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \equiv \langle \langle \delta \theta \rangle \rangle (\varphi_1 U \varphi_2) \]

**Proof.** We prove the equivalent claim that for any \( w \in W \) with \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), any \( z \in Z \), and any variable map \( \theta \),

\[ w, z \models \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \theta \text{ iff } w, z \models \langle \langle \delta \theta \rangle \rangle (\varphi_1 U \varphi_2 \theta) \]

Again note that \( \theta \) is needed since \( \delta, \varphi_1 \) and \( \varphi_2 \) as well as \( \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \) may contain free variables, which by convention we regard as being implicitly \( \forall \)-quantified from the outside.

Let us again first consider the simple case where

\[ \text{LABEL}[G_\delta, \top] \equiv \text{LABEL}[G_\delta, \varphi_2] \text{ AND \text{PATH}[G_\delta]}, \]

i.e. the conjunction of \( \varphi_2 \) with each node’s path existence condition is valid. The procedure then does not enter the while loop, but immediately returns \( \top \) as result, and of course \( \top \theta \) is
trivially satisfied by $w$ in $z$ for any $\theta$. Similarly, by Lemma 5.37 we have that for every $\theta$, there is an infinite run

$$
\langle v_0, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G_\delta} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G_\delta} \ldots
$$

where $w, z \models \varphi_2 \theta$. By Lemma 5.24, $\langle v_i, \theta_i, w, z \cdot p_1 \cdot p_2 \ldots \rangle \not\in \mathcal{F}^{G_\delta}$ for all $i \geq 0$, hence $\pi = p_1 \cdot p_2 \cdot p_3 \cdots \in [G_\delta]_w(\theta)$ by Definition 5.15, therefore also $\pi \in [\delta \theta]_w^{\Sigma}$ by Theorem 5.21, and thus $w, z \models \langle \delta \theta \rangle (\varphi_1 \theta \cup \varphi_2 \theta)$.

Therefore assume now that (5.64) is not the case and that at least one iteration of the while loop is performed. This means that the procedure iteratively constructs labels sets according to below equations:

(5.65) \[ L_0 = \text{LABEL}[G_\delta, \varphi_2] \text{ AND } \text{PATH}[G_\delta] \]

(5.66) \[ L_{i+1} = L_i \text{ OR (LABEL}[G_\delta, \varphi_1] \text{ AND PRE}[G_\delta, L_i]) \]

Semantically, this means that

(5.67) \[ ||L_0||_{\Sigma} = \{ (v, \theta, w, z) \mid w, z \models \varphi_2 \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \} \cap ||\text{PATH}[G_\delta]||_{\Sigma} \]

(5.68) \[ ||L_{i+1}||_{\Sigma} = ||L_i||_{\Sigma} \cup \{ (v, \theta, w, z) \mid w, z \models \varphi_1 \theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \} \cap ||\text{PRE}[G_\delta, L_i]||_{\Sigma} \]

By assumption, the procedure terminates, hence there is some smallest index $n$ such that $L_{n+1} \equiv L_n$. Again it is obvious that the final output has to be a fluent formula. The proof of the main claim of this theorem relies on the following property:

(5.69) \[ \langle v, \theta, w, z \rangle \in ||L_n||_{\Sigma} \text{ iff } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run } \]

$$
\langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G_\delta} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G_\delta} \ldots
$$

such that for some $j \geq 0$, $w, z \cdot p_1 \cdots p_j \models \varphi_2 \theta$

and for all $0 \leq k < j$, $w, z \cdot p_1 \cdots p_k \models \varphi_1 \theta$

Then we get that

$w, z \models \text{CHECKEU}[\delta, \varphi_1, \varphi_2] \theta$

iff $w, z \models \text{INITLABEL}[G_\delta, L_{n+1}] \theta$ \hfill (by assumption)

iff $\langle v_0, \psi_0 \rangle \in L_{n+1}$ and $w, z \models \psi_0 \theta$ \hfill (by (5.46))

iff $\langle v_0, \theta, w, z \rangle \in ||L_{n+1}||_{\Sigma}$ \hfill (by (5.41))

iff $\langle v_0, \theta, w, z \rangle \in ||L_n||_{\Sigma}$ \hfill (since $L_{n+1} \equiv L_n$)
In order to prove (5.69), we introduce a bit of notation for the fact that there exists an infinite run starting in some \(\langle v, \theta, w, z \rangle\) where \(\varphi_2\theta\) becomes true after exactly \(j\) transitions and \(\varphi_1\theta\) holds until then: Let

\[
\langle v, \theta, w, z \rangle \in P(j) \overset{\text{def}}{=} w |\Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run}
\]

\[
\langle v, \theta, w, z \rangle \overset{G_\delta}{\rightarrow} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \overset{G_\delta}{\rightarrow} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \overset{G_\delta}{\rightarrow} \ldots
\]

such that \(w, z \cdot p_1 \ldots p_j \models \varphi_2\theta\)

and for all \(0 \leq k < j\), \(w, z \cdot p_1 \ldots p_k \models \varphi_1\theta\)

and for all \(i \geq 0\), \(\langle v_i, \theta_i, w, z \cdot p_1 \ldots p_i \rangle \not\in F_{G_\delta}\) (by (5.69) and Lemma 5.24)

\[
\text{iff } \pi \in ||G_\delta|| w (\theta) \text{ with } w, z, \pi \models (\varphi_1\theta \cup \varphi_2\theta) \text{ (by Definition 5.15 and the semantics)}
\]

\[
\text{iff } \pi \in ||\delta||^w \text{ with } w, z, \pi \models (\varphi_1\theta \cup \varphi_2\theta) \text{ (by Theorem 5.21)}
\]

\[
\text{iff } w, z \models \langle \langle \delta \theta \rangle \rangle (\varphi_1\theta \cup \varphi_2\theta) \text{ (by the semantics)}
\]

We first show that the following weaker property holds for all \(i \geq 0\):

\[
\langle v, \theta, w, z \rangle \in ||L_i|| \text{ iff for some } j \geq 0, \langle v, \theta, w, z \rangle \in P(j)\]

(5.71)

To prove the above, note that

\[
\text{iff } \langle v, \theta, w, z \rangle \overset{G_\delta}{\rightarrow} \langle v', \theta', w, z \cdot p \rangle, \text{ then } \varphi_1\theta = \varphi_1\theta' \text{ and } \varphi_2\theta = \varphi_2\theta'.
\]

(5.72)

This is again due to the fact that \(v \overset{\phi_1/\pi \times \sigma/\phi_2}{\rightarrow} v'\) is an edge in \(G_\delta\) such that \(\theta' = \theta[\vec{\pi}/\vec{\sigma}]\). Since variables that appear \(\pi\)-quantified in \(\delta\) are assumed to not appear freely in \(\varphi_1\) and \(\varphi_2\), applying \(\theta\) or \(\theta'\) yields the same result. We prove (5.71) by induction on \(i\):

- \(i = 0\): immediate by (5.67) and Lemma 5.37.
- \(i \rightarrow i + 1\):

\[
\langle v, \theta, w, z \rangle \in ||L_{i+1}|| \text{ iff } \langle v, \theta, w, z \rangle \in ||L_i|| \text{ or }
\]

\[
w, z \models \varphi_1\theta, \text{ } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and } \langle v, \theta, w, z \rangle \in ||\text{PRE}[G_\delta, L_i]|| \text{ (by (5.68))}
\]
iff \( \langle v, \theta, w, z \rangle \in \|L_i\|_\Sigma \) or
\[ w, z \models \varphi_1 \theta, \ w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and} \]
\( \langle v, \theta, w, z \rangle \xrightarrow{\bar{G}} (v', \theta', w, z \cdot p) \text{ where} \ (v', \theta', w, z \cdot p) \in \|L_i\|_\Sigma \) (by Lemma 5.32)
iff for some \( j \) with \( i \geq j \geq 0 \), \( \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \) or
\[ w, z \models \varphi_1 \theta, \ w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and} \]
\( \langle v, \theta, w, z \rangle \xrightarrow{\bar{G}} (v', \theta', w, z \cdot p) \)
where for some \( j \) with \( i \geq j \geq 0 \), \( \langle v', \theta', w, z \cdot p \rangle \in \mathcal{P}(j) \) (by induction)
iff for some \( j \) with \( i \geq j \geq 0 \), \( \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \) or
for some \( j \) with \( i + 1 \geq j \geq 1 \), \( \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \) (by (5.70) and (5.72))
iff for some \( j \) with \( i + 1 \geq j \geq 0 \), \( \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \)

Finally, we obtain (5.69) from (5.71) as follows. Assume that \( n \) is the smallest index with \( L_{n+1} \equiv L_n \), but that the algorithm applies (5.68) indefinitely. Then \( \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \) iff \( j \leq n \) and \( \langle v, \theta, w, z \rangle \in \|L_n\|_\Sigma \) by (5.71), or \( j > n \) and \( \langle v, \theta, w, z \rangle \in \|L_j\|_\Sigma \) by (5.71), but where \( \|L_j\|_\Sigma = \|L_n\|_\Sigma \) due to the fact that \( \|L_m\|_\Sigma = \|L_n\|_\Sigma \) for all \( m > n \), which follows inductively from \( L_{n+1} \equiv L_n \).

5.3.3 Correctness

Soundness

The soundness of the subprocedures \textsc{CheckEX}, \textsc{CheckEG} and \textsc{CheckEU} was already established in Theorems 5.34, 5.36 and 5.39, respectively. We are left with showing that the overall \( \mathcal{E} \mathcal{S}_\text{CTL} \) verification transform operator \( \mathcal{C} \) given in Definition 5.26 is sound as well.

Lemma 5.40. Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over \( \langle D, F \rangle \) and \( \varphi \) be a formula of \( \mathcal{E} \mathcal{S}_\text{CTL} \) that only mentions fluents in \( \langle D, F \rangle \). Then
\[ \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \varphi \equiv \mathcal{C}[\varphi] \] (5.73)

Proof. We prove the equivalent claim that if \( w \) is a world with \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \), \( z \) is a sequence of action standard names, and \( \theta \) a variable map,
\[ w, z \models \varphi \theta \text{ iff } w, z \models \mathcal{C}[\varphi] \theta \] (5.74)

The base cases \( \varphi = (t_1 = t_2) \) and \( \varphi = F(i) \) are due to the correctness of regression (Lemma 3.32 (4)) and the easily provable fact that \( R[\alpha \theta] = R[\alpha] \theta \) for any \( \alpha \). Moreover, the cases \( \varphi = \varphi_1 \land \varphi_2, \varphi = \neg \varphi_1 \) and \( \varphi = \exists x \varphi_1 \) are immediately obtained by induction. Cases \( \varphi = (\langle \delta \rangle) X \varphi_1, \)
\[ \varphi = \langle \delta \rangle G \varphi_1 \] and \[ \varphi = \langle \delta \rangle (\varphi_1 U \varphi_2) \] finally follow by induction and using Theorems 5.34, 5.36 and 5.39, respectively.

**Theorem 5.41.** Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over fluents \( (D, F) \) and \( \varphi \) be a sentence of \( \mathcal{ES}_\text{CTL} \) that only mentions fluents in \( D \cup F \). If the computation of \( C[\varphi] \) wrt \( \Sigma \) terminates, it is a fluent sentence and

\[ \Sigma \models \varphi \iff \Sigma_0 \models C[\varphi]. \]

**Proof.** “\( \Rightarrow \)” Suppose \( \Sigma \models \varphi \). Let \( w \) be a world such that \( w \models \Sigma_0 \). From Lemma 3.32 (2) it follows that \( w_{\Sigma} \models \Sigma \), and hence \( w_{\Sigma} \models \varphi \). By Lemma 5.40 we obtain that \( w_{\Sigma} \models C[\varphi] \) and thus by Lemma 3.32 (6) we get \( w \models C[\varphi] \).

“\( \Leftarrow \)” Conversely, suppose \( \Sigma_0 \models C[\varphi] \) and let \( w \) be a world with \( w \models \Sigma \). Then in particular \( w \models \Sigma_0 \), and so \( w \models C[\varphi] \). By Lemma 5.40 \( w \models \varphi \).

**Completeness**

We now know that whenever our algorithm returns an output, then that formula correctly captures precisely the situations in which the input \( \mathcal{ES}_\text{CTL} \) formula is entailed by our basic action theory. The question now is whether the method produces an output for every input.

The quick answer to this question is that obviously, since the algorithm involves classical first-order theorem proving, due to the undecidability of first-order logic, the method necessarily has to be incomplete. Recall from Theorem 3.13 that if \( \alpha \) is a sentence of \( \mathcal{L} \) (which corresponds to the static, objective subset of \( \mathcal{ES} \) without actions and numbers), then \( \alpha \) is valid in FOL iff it is valid in \( \mathcal{ES} \). We can reduce the problem of validity in FOL to the application of our algorithm as follows: Let the basic action theory \( \Sigma \) be the empty set, i.e. we do not make any assumptions about the initial situation and do not have any definitional or normal fluents. Furthermore, suppose \( \delta \) is the program any \( \omega \), which admits every infinite action sequence as trace. Let \( \alpha \) be an \( \mathcal{L} \) sentence that without loss of generality mentions at most rigid predicates and functions, which implies that the truth value of \( \alpha \) does not change by the application of actions. Then \( \alpha \) is valid iff \( \Sigma \models \langle \delta \rangle (\top U \alpha) \).

The above is a serious concern, but it is not specific to our algorithm: In fact, any system that allows for the full expressiveness of first-order logic will necessarily suffer from the logic’s undecidability. In this respect, our method is thus not “less complete” than the overall system that it is part of.

In the spirit of our previous assumption of the agent being a logically omniscient first-order reasoner, let us therefore now assume that we have an oracle that decides all first-order
entailments. A second potential reason for the algorithm failing to terminate is that the loop
condition “$L \not\equiv L$” in line 3 of Procedure 2, line 3 of Procedure 3 or line 3 of Procedure 4 may
never seize to hold.

In fact, it is rather simple to come up with an example where the algorithm does not
terminate. The following is adapted from a similar example in [KP10]. Consider a basic action
theory over a single normal relational fluent $F$ with the successor state axiom

\begin{equation}
\Box[a]F(x) \equiv F(f(x))
\end{equation}

where $f$ is a non-changing function. Again, let $\delta$ be the program any$^\omega$. Suppose we want to
verify whether $\langle\langle \delta \rangle\rangle G F(n)$ for some standard name $n$. Then Procedure 2 iteratively produces
the following label sets:

$L_0 = \{ (v_0, F(n)) \}$
$n_1 = \{ (v_0, F(n) \land F(f(n))) \}$
$L_2 = \{ (v_0, F(n) \land F(f(n)) \land F(f(f(n)))) \}$
\[ \ldots \]
$L_n = \{ (v_0, \bigwedge_{i=0}^{n} F(f^i(n))) \}$

Obviously, the property is true just in case that $F$ holds for all individuals that are the result
of successively applying $f$ to $n$. The latter property is nothing else than a transitive closure,
and hence cannot be expressed by a fluent first-order formula.

The problem here is however not only a matter of expressive power. If we think of $f$ as the
successor function for natural numbers and $n$ as the number zero, then the transitive closure is
the “greater than” relation. Since we defined the meaning of “$>$” within our logic’s semantics,
the property in question can in this case be expressed as follows:

\begin{equation}
F(0) \land \forall m. (m > 0) \supset F(m)
\end{equation}

Yet, our algorithm will fail to find this representation as it cannot be obtained by finitely often
conjoining the current label set with its preimage.

For better understanding under what circumstances we can expect our algorithm to termi-
nate, we have to delve a bit into fixpoint theory, for which we first need the following definition:

**Definition 5.42 (Complete Lattice).** A partially ordered set $\langle L, \leq \rangle$ is a **complete lattice** iff
every subset $A \subseteq L$ has a greatest lower bound, called the *infimum*, or *meet*, and a least upper
bound, called the *supremum*, or *join* in $L$. 
The most prominent example for a complete lattice is the power set \( \mathcal{P}(S) \) of a given set \( S \), for example the set of all graph configurations over \( G_\delta \) wrt \( \Sigma_{\text{det}} \cup \Sigma_{\text{post}} \). In this case, the ordering relation \( \leq \) is set inclusion \( \subseteq \), and the meet and join of some set \( A \) correspond to the union and intersection of all sets in \( A \), respectively. We can then make use of the well-known Knaster-Tarski Theorem:

**Theorem 5.43** (Knaster and Tarski, [Kna28, Tar55]). Let \( (L, \leq) \) be a complete lattice and let \( f : L \to L \) be an order-preserving function (i.e. \( x \leq y \) implies \( f(x) \leq f(y) \)). Then the set of fixpoints of \( f \) (i.e. solutions \( x \in L \) to the equation \( f(x) = x \)) in \( L \) is a complete lattice as well.

In case of Procedure 2 for CheckEG (and similarly Procedure 3 for Path), the monotone function is the semantic counterpart of conjoining a set of labels with its preimage (line 5):

\[
\|L\|_\Sigma = \|L'\|_\Sigma \cap \|\text{PRE}[G_\delta, L']\|_\Sigma
\]

Similarly, the order-preserving operator in case of Procedure 4 for CheckEU is given by

\[
\|L\|_\Sigma = \|L'\|_\Sigma \cup (\|\text{LABEL}[G_\delta, \varphi_1]\|_\Sigma \cap \|\text{PRE}[G_\delta, L']\|_\Sigma)
\]

The good news is that due to Definition 5.42, complete lattices cannot be empty, and therefore the theorem in particular implies the existence of a least and a greatest fixpoint. Moreover, the constructive proof of the theorem [CC79] tells us that these fixpoints can be reached through transfinite iteration. CheckEG and Path hence correspond to iterative approximations of the greatest fixpoint of (5.77) “from above”, while CheckEU iteratively approximates the least fixpoint of (5.78) “from below”.

The bad news on the other hand is that in general, the fixpoint is not necessarily reached within a finite number of iterations. We may rather have to resort to proper transfinite iteration, which may also include the necessity to determine limits of iteration sequences, such as formula (5.76) in the above example. Devising algorithms that are capable of that is however beyond the scope of this thesis and therefore left for future work.

A more direct approach to ensure termination is to restrict oneself to well-founded lattices. Since the latter are defined to not contain any infinite descending chains, the fixpoint computation loop will finish within a finite number of iterations. This could be achieved through appropriately restricting the classes of basic action theories, programs and properties under consideration. A simple, though not very interesting example would be the set of BATs corresponding to the ADL fragment of PDDL (cf. Section 4.2). Since in that case, fluents and actions are restricted to only take arguments from finite domains, the corresponding state space is also finite. In a similar vein, termination can be guaranteed if we restrict ourselves to theories with context-free successor state axioms and a finite number of parameterless actions [LL98b].
Finding more expressive (and interesting) classes of action theories is another worthwhile line of future research. There are at least two approaches to follow: First, we could consider fragments of the Situation Calculus in which the projection problem is known to be decidable [GS10] and study how this affects our algorithm’s termination behaviour, possibly in combination with additional restrictions on action theories, programs and properties. A first step in this direction has been taken by Baader, Liu and ul Mehdi [BLuM10] who use an action formalism based on the decidable description logic $\mathcal{ALC}$ to verify LTL properties, where instead of GOLOG execution traces they consider infinite sequences of actions that are accepted by some Büchi automaton.

Second, it seems promising to build upon results by Liu, Lakemeyer and Levesque [LLL04, LL05b] who present a classically sound, yet incomplete reasoning mechanism that is decidable and even tractable for proper $^+$ knowledge bases. The latter allow to retain a major part of first-order expressiveness, and giving up logical completeness seems a price worth paying in view of the fact that our original algorithm is incomplete anyway. In particular, the approach could be combined with recent results on the tractability of progressing proper $^+$ knowledge bases with respect to local-effect successor state axioms [LL09b], which of course requires redesigning the algorithm such that progression is used instead of regression.

5.4 Expressive Properties of Nonterminating Programs

We have now a good understanding of the verification of properties of non-terminating GOLOG programs, albeit only for a limited class of such properties. Recall the two example questions about the coffee delivery robot’s control program that were given in the beginning of this chapter:

1. Will every request eventually be served by the robot?
2. Is it possible that no request is ever served at all?

We could express both of them in terms of $\mathcal{ESG}$ formulas:

\begin{align*}
(5.79) & \quad [\delta]\mathcal{G}(\text{Occ(requestCoffee}(x)) \supset \text{FOcc}(\text{selectRequest}(x))) \\
(5.80) & \quad \langle\langle\delta\rangle\rangle\mathcal{G} \neg \exists x(\text{Occ}(\text{selectRequest}(x)))
\end{align*}

However, only (5.80), the formula for the second property above, was of a form that the presented algorithm could handle. The $\mathcal{ESG}_{\text{CTL}}$ fragment that we considered resembled the classic temporal logic CTL in that it was required that a $[\delta]$ or $\langle\langle\delta\rangle\rangle$ path quantifier is directly followed by
a temporal operator like $G$ or $F$. In particular, this forbids nesting temporal operators or combining them through logical connectives, such as done in (5.79).

The latter is a representative for a much broader class of properties that resembles classical $\text{CTL}^*$, where the usage of temporal operators within a path quantifier is unrestricted and we may make free use of logical connectives, in our case including first-order quantification. Many of the typical properties one may want to verify for a non-terminating agent are only expressible in this manner, most importantly liveness and fairness conditions such as the above “every request will eventually be served”, or “when the battery is low, it will get recharged in time”, or “the floor gets cleaned infinitely often” etc.

In this section, we therefore address the verification for this more general class of properties, which will be called $\text{ESG}_{\text{CTL}^*}$. The corresponding algorithm will again rely on characteristic program graphs and regression-based reasoning as its main ingredients. As before, there is of course no free lunch here: Allowing arbitrary first-order quantification within action theories, programs and properties, and thus resorting to first-order theorem proving, comes at the price of losing decidability. The algorithm we discuss here is hence sound, but not guaranteed to terminate.

5.4.1 The Logic $\text{ESG}_{\text{CTL}^*}$

First of all we will again formally define the logic under consideration, which is another, more general fragment of $\text{ESG}$.

**Definition 5.44 ($\text{ESG}_{\text{CTL}^*}$).** The formulas of $\text{ESG}_{\text{CTL}^*}$ are the ones admitted by the following grammar:

$$\alpha ::= (t_1 = t_2) \mid F(\vec{t}) \mid \neg \alpha \mid \alpha \land \alpha \mid \exists x.\alpha \mid \langle\langle \delta \rangle\rangle \varphi$$

(5.81)

where temporal subformulas are as follows:

$$\varphi ::= \alpha \mid \neg \varphi \mid \varphi \land \varphi \mid \exists x.\varphi \mid X \varphi \mid \varphi U \varphi$$

(5.82)

Again, we require that all programs $\delta$ are of the form:

$$\delta_1^\omega \parallel \cdots \parallel \delta_k^\omega.$$  

(5.83)

5.4.2 The Algorithm

Our method for verifying $\text{ESG}_{\text{CTL}^*}$ formulas is inspired by the classical propositional $\text{CTL}^*$ model checking algorithm, which in turn relies on a verification method for $\text{LTL}$ formulas
There, the rough idea is to construct a finite, nondeterministic Büchi automaton that encodes the infinite traces not admitted by the input formula. Given the finite transition system that represents the system behaviour, the two models are then merged into a single one. It has then only to be checked whether the resulting transition systems admits any traces at all, which is a purely graph-theoretical task.

First, we need a few definitions. Similar to the $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ case, we have:

**Definition 5.45 (Verifiable $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ Formulas).** Let $\Sigma$ be a basic action theory over $\langle D, F \rangle$. An $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ formula $\alpha$ is verifiable iff it only mentions fluents from $\langle D, F \rangle$, every quantifier uses a distinct variable and every occurrence of a functional fluent is of the form $f(\vec{t}) = \vec{t}'$, where $\vec{t}$ and $\vec{t}'$ do not contain any further functional fluents.

Next, we define a transform operator that recursively evaluates $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ formulas as follows:

**Definition 5.46 (Verification Transform Operator for $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ Formulas).** Let $\Sigma$ be a basic action theory over $\langle D, F \rangle$ and $\alpha$ a verifiable $\mathcal{E}S\mathcal{G}_{\mathcal{S}G^L}$ formula. The verification transformation of $\alpha$ wrt $\Sigma$, denoted as $C[\alpha]$, is inductively defined as follows:

1. $C[(t_1 = t_2)] = \mathcal{R}[(t_1 = t_2)];$
2. $C[F(\vec{t})] = \mathcal{R}[F(\vec{t})];$
3. $C[\neg \alpha] = \neg C[\alpha];$
4. $C[\alpha_1 \land \alpha_2] = C[\alpha_1] \land C[\alpha_2];$
5. $C[\exists x. \alpha] = \exists x.C[\alpha];$
6. $C[\langle\langle \delta \rangle\rangle \varphi] = \text{CHECK}\mathcal{L}\mathcal{T}\mathcal{L}[^{\delta},T[\varphi]].$

Trace subformulas are handled by means of the $T$ operator:

1. $T[\alpha] = C[\alpha]$, if $\alpha$ is a situation formula;
2. $T[\neg \varphi] = \neg T[\varphi];$
3. $T[\varphi_1 \land \varphi_2] = T[\varphi_1] \land T[\varphi_2];$
4. $T[\exists x. \varphi] = \exists x.T[\varphi];$
5. $T[X \varphi] = X T[\varphi];$
6. $T[\varphi_1 U \varphi_2] = T[\varphi_1] U T[\varphi_2];$
We are left with defining CheckLTL. The name of the operation already suggests that it involves evaluating LTL-like formulas. More precisely, we let $\mathcal{ESG}_{LTL}$ refer to the fragment of $\mathcal{ESG}$ trace formulas that do not contain nested path quantifiers, as given by the following grammar:

$$
\varphi ::= (t_1 = t_2) \mid F(\vec{t}) \mid \neg \varphi \mid \varphi \land \varphi \mid \exists x. \varphi \mid X \varphi \mid \varphi U \varphi
$$

(5.84)

It is easy to see that any input formula to CheckLTL $[\delta, \cdot]$ during the evaluation of $C[\alpha]$ has to be of this form, provided that CheckLTL itself only yields fluent formulas as result (which will be shown later).

In the classical automata-based LTL model checking approach, the states of the automaton represent the elementary sets of subformulas in the input formula $\varphi$: Given all subformulas in the input formula $\varphi$, together with their negations, a subset is called elementary iff it is logically consistent, the satisfaction of temporal subformulas is locally consistent, and the set is maximal. The transitions in the automaton are furthermore defined according to the semantics of the logic and the expansion laws of temporal operators. The acceptance criterion is chosen in a way such that it is ensured that any accepted trace satisfies $\varphi$.

Unfortunately, it is not possible to directly adopt this approach for the verification of $\mathcal{ESG}_{LTL}$ formulas. The reason is that we allow arbitrary first-order quantification inside the temporal input formula, and hence the set of its subformulas is in general not finite. We hence have to again resort to an implicit representation using logical formulas.

Formally, let $\varphi \in \mathcal{ESG}_{LTL}$. We need to consider all (and only) temporal subformulas of $\varphi$, i.e. all its subformulas of the form $(\phi_i U \psi_i)$ (with free variables $\vec{x}_i$) as well as all subformulas of the form $X \phi_j$ (with free variables $\vec{x}_j$).

**Example 5.47.** Consider property (5.79) that expresses that every request will eventually be served. Removing all “syntactic sugar”, the formula reads as follows:

$$
\forall x. \neg \langle\langle \delta \rangle\rangle (\top U (Occ(rC(x)) \land \neg (\top U Occ(sR(x)))))
$$

(5.85)

Let $\varphi$ denote the part behind the $\langle\langle \delta \rangle\rangle$ quantifier. It is an $\mathcal{ESG}_{LTL}$ formula and contains the following two temporal subformulas with free variable $x$:

$$
\top U Occ(sR(x))
$$

(5.86)

$$
\top U (Occ(rC(x)) \land \neg (\top U Occ(sR(x))))
$$

(5.87)

In a similar vein as LTL model checking with Büchi automata, the idea behind our approach is now to break down the truth of temporal formulas on infinite traces into three different parts:
1. **Local Consistency:**

First, we have to ensure the local consistency of satisfaction of the temporal subformulas:

\[(5.88) \quad \forall \vec{x}_i. \psi_i \supset (\phi_i \ U \psi_i)\]

\[(5.89) \quad \forall \vec{x}_i. (\phi_i \ U \psi_i) \land \neg \psi_i \supset \phi_i\]

(5.88) expresses that any trace starting in the current situation, where \( \psi_i \) is satisfied, also satisfies \((\phi_i \ U \psi_i)\). Moreover, if \( \psi_i \) does no hold, then such a trace can only satisfy \((\phi_i \ U \psi_i)\) if \( \phi_i \) is true (5.89).

Note that due to our implicit representation, it is unnecessary to explicitly enforce logical consistency and maximality, since we get these “for free” from logical entailment: If a label formula is inconsistent, then the set of satisfying configurations will be empty, and the set of formulas entailed by some label formula is maximal by definition.

2. **Single-Step Consistency:**

Next, it is required that the satisfaction of temporal subformulas remains consistent from one situation of a trace to the next:

\[(5.90) \quad \forall \vec{x}_i. (\phi_i \ U \psi_i) \equiv \psi_i \lor (\phi_i \land [a](\phi_i \ U \psi_i))\]

\[(5.91) \quad \forall \vec{x}_j. X \phi_j \equiv [a] \phi_j\]

(5.90) is a variant of the well-known expansion law for the temporal “until” operator, whereas (5.91) is a similar one for “next”.

3. **Eventual Compliance:**

Finally, for each \((\phi_i \ U \psi_i)\) subformula, we have to ensure that the acceptance condition

\[(5.92) \quad Accept_i \overset{\text{def}}{=} (\phi_i \ U \psi_i) \supset \psi_i\]

gets satisfied infinitely often.

**Definition 5.48.** Given \( \varphi \in ESG_{\text{LTL}} \), we let \( \text{LocCons}[\varphi] \) denote the set of all local consistency constraints of the forms (5.88) and (5.89) for all of \( \varphi \)'s temporal “until” subformulas. Similarly, we will use \( \text{Trans}[\varphi] \) to denote the set of all transition constraints (5.90) and (5.91) for temporal subformulas in \( \varphi \).

Moreover, in order to be able to reason with these properties solely based on non-modal theorem proving, we introduce a new predicate symbol \( U_i(\vec{x}_i) \) for each \((\phi_i \ U \psi_i)\) subformula (whose free variables are \( \vec{x}_i \)), and a new predicate symbol \( X_j(\vec{x}_j) \) for each \( X \phi_j \) subformula (whose free variables are \( \vec{x}_j \)). We will use \( \phi \downarrow \) to denote the result of replacing all of \( \varphi \)'s temporal subformulas in \( \phi \) by their corresponding predicates.
Example 5.49. For the two temporal subformulas mentioned in Example 5.47 we have to introduce two new predicate symbols. Let $U_1(x)$ correspond to temporal subformula (5.86), and $U_2(x)$ to (5.87). Then $LocCons[\varphi] \downarrow$ contains

\begin{align*}
&\forall x. \text{Occ}(sR(x)) \supset U_1(x) \\
&(5.93) \\
&\forall x. U_1(x) \land \neg \text{Occ}(sR(x)) \supset \top \\
&(5.94) \\
&\forall x. \text{Occ}(rC(x)) \land \neg U_1(x) \supset U_2(x) \\
&(5.95) \\
&\forall x. U_2(x) \land \neg (\text{Occ}(rC(x)) \land \neg U_1(x)) \supset \top \\
&(5.96)
\end{align*}

Conveniently, formulas (5.94) and (5.96) are valid, hence we will only have to deal with (5.93) and (5.95) in the following.

Next, $Trans[\varphi] \downarrow$ in the example is the set consisting of the following formulas:

\begin{align*}
&\forall x. U_1(x) \equiv \text{Occ}(sR(x)) \lor [a] U_1(x) \\
&(5.97) \\
&\forall x. U_2(x) \equiv \text{Occ}(rC(x)) \land \neg U_1(x) \lor [a] U_2(x) \\
&(5.98)
\end{align*}

Finally, the acceptance conditions are

\begin{align*}
&\text{Accept}_1 \downarrow = U_1(x) \supset \text{Occ}(sR(x)) \\
&(5.99) \\
&\text{Accept}_2 \downarrow = U_2(x) \supset \text{Occ}(rC(x)) \land \neg U_1(x) \\
&(5.100)
\end{align*}

To ensure eventual compliance, we use a property that is similar to the acceptance criterion of Büchi automata, but extends it to the first-order case. For this purpose, we introduce yet another new fluent $A_i(\vec{x}_i)$ for each $(\phi_i U \psi_i)$ subformula, having the successor state axiom

\begin{align*}
&\square[a] A_i(\vec{x}_i) \equiv (\text{Accept}_i \downarrow \lor (A_i(\vec{x}_i) \land \neg \text{AccAll}[\varphi])) \\
&(5.101)
\end{align*}

where

\begin{align*}
&\text{AccAll}[\varphi] \overset{\text{def}}{=} \bigwedge_i \forall \vec{x}_i. A_i(\vec{x}_i) \\
&(5.102)
\end{align*}

The idea is that the $A_i$ “collect” all instances of $\text{Accept}_i$, and $\text{AccAll}[\varphi]$ becomes true once all $A_i$ hold for all $\vec{x}_i$, after which they are “reset” to false. The algorithm then basically tries to prove the existence of an execution trace on which $\text{AccAll}[\varphi]$ is satisfied infinitely often. The algorithm is depicted below:
5.4 Expressive Properties of Nonterminating Programs

Procedure 5 \textsc{CheckLTL}[\delta, \varphi]

\begin{enumerate}
\item \(L' := \text{Label}[G_{\delta}, \bot]\);
\item \(L := \text{Label}[G_{\delta}, \text{AccAll}[\varphi]] \text{ AND PATHLTL}[G_{\delta}, \varphi];\)
\item \textbf{while} \(L \neq L'\) \textbf{do}
\item \(L' := L;\)
\item \(L := L' \text{ OR PRELTL}[G_{\delta}, \varphi, L'];\)
\item \textbf{end while}
\item \(L' := \text{Label}[G_{\delta}, \bot];\)
\item \textbf{while} \(L \neq L'\) \textbf{do}
\item \(L' := L;\)
\item \(L := L' \text{ AND PRELTL}[G_{\delta}, \varphi, L'];\)
\item \textbf{end while}
\item \textbf{return} \textsc{InitLabelLTL}[G_{\delta}, \varphi, L]
\end{enumerate}

Intuitively, we can understand its structure as follows: In line 2, we initialize the label set to represent all configurations on situations where \(\text{AccAll}[\varphi]\) holds and that may start infinite runs. The fixpoint computation loop in lines 3 to 6 then determines the configurations from which \textit{eventually} such configurations are reachable (note the similarity to Procedure 4 for \textsc{CheckEU}). Furthermore, the loop in lines 8 to 11 computes labels representing configurations where this is \textit{always} the case (again note the resemblance to Procedure 2 for \textsc{CheckEG}). The formula extracted in line 12 hence represents the necessary and sufficient conditions under which there is a path where \(\text{AccAll}[\varphi]\) is \textit{always eventually} true, i.e. holds infinitely often.

A few details are left to explain. Foremost, we have to revise the definition of a preimage, as it is now required that we ensure that local and single-step consistency is maintained:

\textbf{Definition 5.50} (Preimage Under Local And Single-Step Constraints). Let \(L\) be a set of labels on some trace graph \(G = \langle V, E, v_0 \rangle\) and \(\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}\) be a basic action theory. The \textit{preimage under local and single-step constraints} of \(L\) is defined as:

\begin{equation}
\text{PRE}_{\text{LTL}}[\langle V, E, v_0 \rangle, \varphi, L] = \{ (v, \text{PRE}_{\text{LTL}}[v, \varphi, L]) \mid v \in V \}
\end{equation}

where the preimage of a single node is

\begin{equation}
\text{PRE}_{\text{LTL}}[v, \varphi, L] = \bigvee \{ \text{PRE}_{\text{LTL}}[\phi, \vec{x}, t, \phi', \psi, \varphi] \mid v \xrightarrow{\phi/\vec{x}/t/\phi'} v' \in E, \langle v', \psi \rangle \in L \}
\end{equation}

with

\begin{equation}
\text{PRE}_{\text{LTL}}[\phi, \vec{x}, t, \phi', \psi, \varphi] = \text{ELIM}[\langle \vec{X}, \vec{U} \rangle, R[\text{LocCons}[\varphi] \downarrow \land \phi \land \exists \vec{x}. \phi' \land [t]\psi \land \text{Trans}[\varphi]\downarrow \pi]].
\end{equation}
The formula to be regressed may now contain occurrences of the newly introduced special fluents. We therefore extend the set of successor state axioms \( \Sigma_{\text{post}} \) by axioms of the form (5.101) for all \( A_i(x_i) \) (in the following denoted by \( \Sigma_A \) post as well as the following ones for \( U_i(x_i) \) and \( X_j(x_j) \) (in the following denoted by \( \Sigma_X, \Sigma_U \) post):

\[
\Box [a] U_i(x_i) \equiv U_i(x_i) \tag{5.106}
\]

\[
\Box [a] X_j(x_j) \equiv X_j(x_j) \tag{5.107}
\]

and regress wrt \( \Sigma_{\text{post}} \cup \Sigma_A \) post \( \cup \Sigma_X, \Sigma_U \) post. The \( U_i \) and \( X_j \) are yet another set of new predicates. Intuitively, they are placeholders for the truth values of the \( U_i \) and \( X_j \) in the successor situation. After regression, we eliminate them using the \( \text{Elim} \) operator:

\[
\text{Elim}[\langle P_1, \ldots, P_k \rangle, \alpha] \overset{\text{def}}{=} \exists P_1 \ldots \exists P_k \alpha \tag{5.108}
\]

We thus get a representation of those configurations for which a truth assignment of the temporal subformulas exists such that local and single-step consistency are maintained into the successor configuration.

The definition of \( \text{Elim} \) given in (5.108) obviously amounts to second-order reasoning, which violates our assumption of a logically omniscient first-order reasoner that we took so far. In Section 5.4.4 below, we briefly discuss how an alternative definition of the operator allows to stay consistent with the assumption. Second-order quantification is necessary because of the nondeterministic nature of the single-step consistency law for the “until” case: as opposed to successor state axioms for normal fluents, (5.90) cannot determine a unique value for \((\phi_i U_i \psi_i)\) (and thus \( U_i \)) in the successor situation solely based on the truth values of formulas in the current situation. We use \( U_i \) therefore rather to express the possibility of that \((\phi_i U_i \psi_i)\) holds on some execution trace, and additionally have to ensure eventual compliance otherwise.

Furthermore, for the above to be well-defined, we require that label formulas are now always fluent formulas wrt \( \langle D, \mathcal{F} \cup \{ \vec{X}, \vec{U}, \vec{A} \} \rangle \). This implies that the \( \vec{X} \) and \( \vec{U} \) that occur in any label formula are always bound by a second-order quantifier. It can easily be verified that the initial assignment of label sets in Procedure 5 conforms to this requirement, and that it will be maintained by the application of \( \text{PRE}_{\text{LTL}} \) as well as conjunction and disjunction of labels.

Let us first understand this new notion of a preimage semantically. For this purpose, we introduce the following relation among worlds:

**Definition 5.51.** If \( w, w' \in \mathcal{W} \), \( z \in \mathcal{Z} \) and \( P \) (fluent or rigid) predicate symbols, then \( w \overset{z}{\approx}_{P} w' \) iff \( w \overset{z'}{\approx}_{P} w' \) and for all \( z' \neq z \cdot z'' \), all primitive terms \( t \) and all primitive sentences \( \beta \):

\[
w[t, z'] = w'[t, z'] \tag{5.109}
\]

\[
w[\beta, z'] = w'[\beta, z'] \tag{5.110}
\]
5.4 Expressive Properties of Nonterminating Programs

Intuitively, \( w \approx_P w' \) means that \( w \) and \( w' \) are almost identical, except for their interpretation of the \( P \) from \( z \) on. Recall that this is different from \( w \approx_P w' \) which allows the two worlds to differ arbitrarily in situations not reachable from \( z \). We note the following consequences of this definition:

**Proposition 5.52.** \( \approx_P \) is an equivalence relation.

**Proposition 5.53.** If \( w' \approx_P w \), then \( w' \approx_P w' \).

**Proposition 5.54.** If \( w' \approx_P w \) and \( w',z \models \alpha \), then there exists \( w'',z \models \alpha \).

Additionally, we need:

**Definition 5.55 (Instantiation of Successor State Axioms).** Let \( \Sigma_{post} \) be any set of SSAs

\[
\Box[a]F(\vec{x}) \equiv \gamma_F.
\]

Then \( \Sigma_{post}(p) \) denotes the instantiation of \( \Sigma_{post} \) wrt \( p \), consisting of corresponding formulas

\[
([a]F(\vec{x}) \equiv \gamma_F)^p.
\]

With the above, we can now define:

**Definition 5.56 (Runs Consistent wrt an \( \mathcal{ESGLTL} \) Formula).** Given a trace graph \( G = \langle V, E, v_0 \rangle \) and a formula \( \varphi \in \mathcal{ESGLTL} \), we extend the definition of transition steps and finality for graph configurations to include consistency wrt \( \varphi \) as follows:

- \( \langle v, \theta, w, z \rangle \xrightarrow{G, \varphi} \langle v', \theta', w', z \cdot p \rangle \) iff \( \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta', w, z \cdot p \rangle \), \( w' \approx_P \vec{\chi}, \theta, \vec{A} \) \( w \) and \( w', z \models \text{LocCons}[\varphi] \land \text{Trans}[\varphi]_{\vec{x}p}^{a} \land \Sigma_{post}^{A}(p) \).

- \( \langle v, \theta, w, z \rangle \in \mathcal{F}^{G,\varphi} \) iff \( \langle v, \theta, w, z \rangle \in \mathcal{F}^{G} \) and \( w, z \models \text{LocCons}[\varphi] \).

Furthermore, the notion of (finite and infinite) runs from Definition 5.15 is extended accordingly.

Note that this new definition of a transition among graph configurations now allows to switch to a different world, albeit one that at most differs in the interpretation of the auxiliary predicates starting at the current situation. Also notice that the formulas \( \text{LocCons}[\varphi] \downarrow \), \( \text{Trans}[\varphi]_{\vec{x}p}^{a} \) and \( \Sigma_{post}(p) \) are actually sentences, hence their satisfaction does not depend on the application of a variable map. We now have, in analogy to Lemma 5.32:
Lemma 5.57. Let $G = (V, E, v_0)$ be a trace graph, $\Sigma$ a basic action theory and $\varphi \in \mathcal{ESLTL}$.

$|\text{PRELTL}[G, \varphi, L]|_{\Sigma} = \{\langle v, \theta, w, z \rangle \mid \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta', w', z \cdot p \rangle, \langle v', \theta', w', z \cdot p \rangle \in L|_{\Sigma}\}.$

Proof.

\begin{align*}
&\langle v, \theta, w, z \rangle \in |\text{PRELTL}[G, \varphi, L]|_{\Sigma} \\
&\text{iff } \langle v, \theta, w, z \rangle \in |\text{PRELTL}[v, \varphi, L]|_{\Sigma} \quad \text{(by (5.103) and (5.42))} \\
&\text{iff } w, z \models \text{PRELTL}[v, \varphi, L]|_{\theta} \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.41))} \\
&\text{iff } w, z \models \exists X, U. \mathcal{R}[\text{LocCons}[^{\varphi} \land \exists \overrightarrow{\alpha} \land [t] \psi \land \text{Trans}[\varphi]^{\varphi}]| U \theta, \\
&\text{where } v \xrightarrow{\phi/\overrightarrow{x} \cdot \overrightarrow{n}} v' \in E, \langle v', \psi \rangle \in L \\
&\text{and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.104) and (5.108))} \\
&\text{iff exists } w' \xrightarrow{\varphi} X, U \theta \text{ w and standard names } \overrightarrow{n} \text{ such that } w', z \models \mathcal{R}[\text{LocCons}[^{\varphi} \land \exists \overrightarrow{\alpha} \land [t] \psi \land \text{Trans}[\varphi]^{\varphi}]| U \theta, \\
&\text{where } v \xrightarrow{\phi/\overrightarrow{x} \cdot \overrightarrow{n}} v' \in E, \langle v', \psi \rangle \in L \\
&\text{and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by the semantics)} \\
&\text{iff exists } w' \xrightarrow{\overrightarrow{p}} X, U \theta \text{ w and standard names } \overrightarrow{n} \text{ such that } w', z \models \phi \land \exists \overrightarrow{x} [\varphi]^{\varphi} [\theta] (\overrightarrow{n}), \\
&\text{w', z \models \text{LocCons}[^{\varphi} \land \exists \overrightarrow{\alpha} \land \text{Trans}[\varphi]^{\varphi} \land \Sigma_{\text{post}}(p)] \text{ and } w', z \cdot p \models \psi [\overrightarrow{x} / \overrightarrow{n}], \\
&\text{where } v \xrightarrow{\phi/\overrightarrow{x} \cdot \overrightarrow{n}} v' \in E, \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.112) below)} \\
&\text{iff exists } w' \xrightarrow{\overrightarrow{p}} X, U \theta \text{ w and standard names } \overrightarrow{n} \text{ such that } z, w \models \phi \land \exists \overrightarrow{x} [\varphi]^{\varphi} (\overrightarrow{n}), \\
&\text{w', z \models \text{LocCons}[^{\varphi} \land \text{Trans}[\varphi]^{\varphi} \land \Sigma_{\text{post}}(p)] \text{ and } w', z \cdot p \models \psi [\overrightarrow{x} / \overrightarrow{n}], \\
&\text{where } v \xrightarrow{\phi/\overrightarrow{x} \cdot \overrightarrow{n}} v' \in E, \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by (5.111) below)} \\
&\text{iff exists } w' \xrightarrow{\overrightarrow{p}} X, U \theta \text{ w and standard names } \overrightarrow{n} \text{ such that } \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta [\overrightarrow{x} / \overrightarrow{n}], w, z \cdot p \rangle, \\
&\text{w', z \models \text{LocCons}[^{\varphi} \land \text{Trans}[\varphi]^{\varphi} \land \Sigma_{\text{post}}(p)] \text{ and } w', z \cdot p \models \psi [\overrightarrow{x} / \overrightarrow{n}], \\
&\text{where } \langle v', \psi \rangle \in L \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by Definition 5.15)} \\
&\text{iff } \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta [\overrightarrow{x} / \overrightarrow{n}], w', z \cdot p \rangle, \\
&\text{w', z \cdot p \models \psi [\overrightarrow{x} / \overrightarrow{n}], \langle v', \psi \rangle \in L, \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \quad \text{(by Definition 5.56)} \\
&\text{iff } \langle v, \theta, w, z \rangle \xrightarrow{G} \langle v', \theta [\overrightarrow{x} / \overrightarrow{n}], w', z \cdot p \rangle \text{ and } \langle v', \theta [\overrightarrow{x} / \overrightarrow{n}], w', z \cdot p \rangle \in |L|_{\Sigma} \quad \text{(by (5.41))}
\end{align*}
In the seventh step above, we used the following property:

\[(5.111) \quad \text{If } w' \not\approx_{X, \vec{U}, \vec{A}} w, \text{ then } w' \text{ and } w \text{ agree on } \phi, \phi', t \text{ and } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}.\]

This is because by assumption, neither of \(\phi, \phi', t \text{ and } \Sigma_{\text{def}} \cup \Sigma_{\text{post}}\) mentions any of the \(\langle \vec{X}, \vec{U}, \vec{A}\rangle\).

It remains to be proven that the sixth rewriting step is correct as well. For this purpose, assume \(w | = \Sigma_{\text{def}} \cup \Sigma_{\text{post}}\), that \(\alpha\) is regressable wrt \(\Sigma_{\text{def}} \cup \Sigma_{\text{post}} \cup \Sigma_{\vec{X}, \vec{U}} \cup \Sigma_{\vec{A}}\), that the only action within \([\cdot]\) operators in \(\alpha\) is \(t\theta\) and that \(\alpha\) does not contain any free occurrences of the \(\vec{X}, \vec{U}\). Then we use that:

\[(5.112) \quad \text{There is } w' \sim_{X, \vec{U}, \vec{A}} w \text{ with } w' | = R[\alpha]z \quad \text{iff} \quad \text{there is } w'' \not\approx_{X, \vec{U}, \vec{A}} w \text{ with } w'', z | = \alpha \theta \land \Sigma_{\vec{A}}(p), \text{ where } p = |t\theta|_w.\]

We prove the two directions separately:

“⇒”: Let \(p = |t\theta|_w\) and \(w''\) be a world such that for all \(\vec{n}\),

\[(5.113) \quad w''[X_j(\vec{n}), z \cdot p] = w'[X_j(\vec{n}), z]\]
\[(5.114) \quad w''[U_i(\vec{n}), z \cdot p] = w'[U_i(\vec{n}), z]\]
\[(5.115) \quad w''[A_i(\vec{n}), z \cdot p] = 1 \iff w'', z | = \gamma A_i \vec{x}_i a_{\vec{n}} p]\]

and that is otherwise like \(w\). Obviously, \(w'' \not\approx_{X, \vec{U}, \vec{A}} w\) and \(w'', z | = \Sigma_{\vec{A}}(p)\). That \(w'', z | = \alpha \theta\) if \(w' | = R[\alpha]z\) can then be shown by induction on \(\alpha\), similar to Lemma 3.32 item 4.

“⇐”: Conversely, given \(w''\), we define \(w'\) to the world that satisfies \(5.113\) and \(5.114\), and that is otherwise like \(w\). Then obviously \(w' \sim_{X, \vec{U}, \vec{A}} w\). Again, \(w'', z | = \alpha \theta\) if \(w' | = R[\alpha]z\) follows by induction on the structure of \(\alpha\).

Next, Procedure 5 uses a variant of the PATH algorithm (Procedure 3):

**Procedure 6 PATH\_LTL[\(\mathcal{G}_\delta, \varphi\)]**

```plaintext
1: L' := Label[\(\mathcal{G}_\delta, \bot\)];
2: L := Label[\(\mathcal{G}_\delta, \top\)];
3: while L \neq L' do
4:    L' := L;
5:    L := L' \text{ AND PRE\_LTL}[\mathcal{G}_\delta, \varphi, L'];
6: end while
7: return L
```

We note that it behaves completely analogous to PATH:
Lemma 5.58. Let $\Sigma$ be a basic action theory and $\delta$ a program of the form $\delta^1\cdot\cdots\cdot\delta^k$ without free variables. If Procedure 6 terminates, it returns a label set $L$ such that

$$\langle v, \theta, w, z \rangle \in \| L \| \Sigma \iff w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and there is an infinite run }$$

$$\langle v, \theta, w, z \rangle \xrightarrow{\delta^1, \varphi} \langle v_1, \theta_1, w_1, z \cdot p_1 \rangle \xrightarrow{\delta^1, \varphi} \langle v_2, \theta_2, w_2, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{\delta^1, \varphi} \cdots$$

Proof. Similar to the proof of Lemma 5.37, only using Lemma 5.57 instead of Lemma 5.32. \(\square\)

The last definition for the algorithm is the extraction of the result formula out of the initial node’s label, which once more requires eliminating auxiliary predicates:

(5.116) \[ \text{INITLABEL}_{\text{LTL}}[G, \varphi, L] \overset{\text{def}}{=} \text{ELIM}[(\bar{X}, \bar{U}, \bar{A}), \varphi \downarrow \land \psi] \text{ such that } \langle v_0, \psi \rangle \in L. \]

5.4.3 Correctness

Before we come to the main theorem of this section, we first establish the following relation between runs with respect to some $\varphi \in E_\text{LTL}$ and runs satisfying $\varphi$:

Lemma 5.59. Let $w \in W$ and $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ a basic action theory over $\langle D, F \rangle$ such that $w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$. Furthermore let $\varphi \in E_\text{LTL}$ and $G = \langle V, E, v_0 \rangle$ a trace graph. The following are equivalent:

1. There is some $w_0 \approx \bar{X}, \bar{U}, \bar{A}$ with $w_0, z \models \varphi \downarrow \theta$ and a run

$$\langle v_0, \theta, w_0, z \rangle \xrightarrow{G, \varphi} \langle v_1, \theta_1, w_1, z \cdot p_1 \rangle \xrightarrow{G, \varphi} \langle v_2, \theta_2, w_2, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G, \varphi} \cdots$$

such that for all $k \geq 0$ exists $l \geq k$ with $w_1, z \cdot p_1 \cdots p_l \models \text{AccAll}[\varphi]$.

2. $w, z, p_1 \cdot p_2 \cdots \models \varphi \land \theta$ and there is a run

$$\langle v_0, \theta, w, z \rangle \xrightarrow{G} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G} \langle v_2, \theta_2, w, z \cdot p_1 \cdot p_2 \rangle \xrightarrow{G} \cdots$$

Proof. We only sketch the structure of the proof here. The details are in Appendix A.3.

"1 $\Rightarrow$ 2": We establish the existence of a corresponding run by an induction over the steps of the given run. The satisfaction of $\varphi \theta$ is shown by induction over the structure of $\varphi$.

"2 $\Rightarrow$ 1": We construct a world $w_0$ that is like $w$, except that for all $k \geq 0$, all $i$, all $j$, and all $\bar{n}$

$$w_0[X_j(\bar{n}), z \cdot z_k] = 1 \iff w, z \cdot z_k, \pi_k \models X^{\varphi_j}_{\bar{n}}$$

$$w_0[U_i(\bar{n}), z \cdot z_k] = 1 \iff w, z \cdot z_k, \pi_k \models (\phi_i U^{\psi_i}_{\bar{n}})^x_{\bar{n}}$$

$$w_0[A_i(\bar{n}), z] = 0$$

$$w_0[A_i(\bar{n}), z \cdot z_{k+1}] = 1 \iff w_0, z \cdot z_k \models \gamma A^{x_j}_{i n} a_{p_{k+1}}$$

To obtain a sequence of worlds $w_0, w_1, w_2, \ldots$ with the desired properties, we set $w_k = w_0$ for all $k > 0$. \(\square\)
With the above lemma, it is now possible to show that Procedure 5 works correctly:

**Theorem 5.60.** Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory over $\langle D, F \rangle$, $\delta$ a program of the form $\delta_1^\omega \| \cdots \| \delta_k^\omega$ that only mentions fluents in $\langle D, F \rangle$, and $\varphi$ a formula of $\mathcal{ESGLTL}$ that only mentions fluents in $\langle D, F \rangle$. If the procedure terminates, $\text{CHECKLTL}[\delta, \varphi]$ is fluent and

$$\Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \text{CHECKLTL}[\delta, \varphi] \equiv \langle \langle \delta \rangle \rangle \varphi$$

**Proof.** We will prove the equivalent claim that for all $w \in \mathcal{W}$ such that $w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$, all $z \in Z$ and all variable maps $\theta$,

$$w, z \models \text{CHECKLTL}[\delta, \varphi] \theta \iff w, z \models \langle \langle \delta \theta \rangle \rangle \varphi \theta.$$

So assume that the procedure terminates. Let $L_0$ denote the label set $L$ after the initial assignment in line 2, $L_1$ be $L$ after completing the first while loop in line 6, and $L_2$ after finishing the second while loop in line 11. Then the following properties hold:

For $w, z \models \text{CHECKLTL}[\delta, \varphi] \theta$ iff $w, z \models \langle \langle \delta \theta \rangle \rangle \varphi \theta$.

The above properties can be proven as follows:

- (5.117): by Lemma 5.58 and our interpretation of label formulas.

- (5.118): Similar to property (5.69) in the proof for Theorem 5.39: Before entering the first while loop in line 3, the set of labels is equivalent to $L_0$. After one iteration, $L$ represents those configurations such that a configuration in $[L_0]_{\Sigma}$ can be reached in one step. After
two iterations, \( L \) stands for those configurations such that a configuration in \( \|L_0\|_\Sigma \) can be reached in at most two steps, and so on. If the loop converges after \( n \) iterations, it means that the resulting \( L \) stands not only for all configurations where a configuration in \( |L_0|_\Sigma \) is reachable within at most \( n \) steps, but also those where this is the case for any number of steps \( l > n \).

- (5.119): Similar to property (5.62) in the proof for Theorem 5.36: Again, before the procedure enters the second while loop in line 8, the set of labels is \( L_1 \). After doing one iteration, \( L \) represents those configurations where it is ensured that there starts a run such that all visited configurations until after the first step are in \( \|L_1\|_\Sigma \). After the second iteration, \( L \) stands for those configurations that start some run where all configurations until the completing the first two transitions are from \( |L_1|_\Sigma \), and so on. If the loop converges after \( n \) iterations, it means that \( L \) represents those configurations where there is a path such that not only the configurations visited within the first \( n \) transitions remain in \( \|L_1\|_\Sigma \), but all of them do.

Now let \( w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \). Then

\[
\begin{align*}
w, z & | \text{CHECKLTL}[\delta, \varphi] \theta \\
\text{iff } w, z & | \text{INITLABEL}_{\text{LTL}}[\delta, \varphi, L_2] \theta \\
\text{iff } w, z & | \exists \vec{X} \cup \vec{A} . (\varphi \downarrow \wedge \psi) \theta \text{ and } \langle v_0, \psi \rangle \in L_2 \quad \text{(by assumption)} \\
\text{iff there is some } w' \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w', z | \varphi \downarrow \theta, w', z | \psi \theta \text{ and } \langle v_0, \psi \rangle \in L_2 \quad \text{(by assumption (5.116))} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \psi \rangle \in L_2 \quad \text{(by the semantics)} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \theta, v_0, z \rangle \in \|L_2\|_\Sigma \quad \text{(by Proposition 5.54)} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \theta, w_0, z \rangle \in \|L_2\|_\Sigma \quad \text{(by (5.41))} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \theta, w_0, z \rangle \in \|L_2\|_\Sigma \quad \text{(by (5.119))} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \theta, w_0, z \rangle \in \|L_2\|_\Sigma \quad \text{(by (5.59))} \\
\text{iff there is some } w_0 \approx \vec{X}, \vec{U}, \vec{A} w \text{ with } w_0, z | \varphi \downarrow \theta, w_0, z | \psi \theta \text{ and } \langle v_0, \theta, w_0, z \rangle \in \|L_2\|_\Sigma \quad \text{(by Lemma 5.24)}
\end{align*}
\]
5.4 Expressive Properties of Nonterminating Programs

Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory over $\langle D, F \rangle$ and $\varphi$ a formula of $\mathcal{ESG}_{\text{CTL}}^*$ that only mentions fluents in $\langle D, F \rangle$. Then

\begin{equation}
\Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \Box \varphi \equiv C[\varphi]
\end{equation}

Proof. We prove the equivalent claim that if $w$ is a world with $w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$, $z$ is a sequence of action standard names, and $\theta$ a variable map,

\begin{equation}
w, z \models \varphi \theta \text{ iff } w, z \models C[\varphi] \theta
\end{equation}

The base cases $\varphi = (t_1 = t_2)$ and $\varphi = F(\vec{t})$ are due to the correctness of regression (Lemma 3.32 (4)) and the easily provable fact that $R[\alpha \theta] = R[\alpha] \theta$ for any $\alpha$. Moreover, the cases $\varphi = \varphi_1 \land \varphi_2$, $\varphi = \neg \varphi_1$ and $\varphi = \exists x \varphi_1$ are immediately obtained by induction. The case $\varphi = \langle \langle \delta \rangle \rangle_{\phi}$ finally follows by an easy induction on the structure of $\phi$ and Theorem 5.60.

Theorem 5.62. Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory over $\langle D, F \rangle$ and $\varphi$ a sentence of $\mathcal{ESG}_{\text{CTL}}^*$ that only mentions fluents in $D \cup F$. If the computation of $C[\varphi]$ wrt $\Sigma$ terminates, it is a fluent sentence and

$$\Sigma \models \varphi \text{ iff } \Sigma_0 \models C[\varphi].$$

Proof. "⇒": Suppose $\Sigma \models \varphi$. Let $w$ be a world such that $w \models \Sigma_0$. From Lemma 3.32 (2) it follows that $w_{\Sigma} \models \Sigma$, and hence $w_{\Sigma} \models \varphi$. By Lemma 5.61 we obtain that $w_{\Sigma} \models C[\varphi]$ and thus by Lemma 3.32 (6) we get $w \models C[\varphi]$.

"⇐": Conversely, suppose $\Sigma_0 \models C[\varphi]$ and let $w$ be a world with $w \models \Sigma$. Then in particular $w \models \Sigma_0$, and so $w \models C[\varphi]$. By Lemma 5.61, $w \models \varphi$.

We thus established the soundness of our verification algorithm for $\mathcal{ESG}_{\text{CTL}}^*$. Regarding its completeness, it is easy to see that everything that we discussed in Section 5.3.3 for $\mathcal{ESG}_{\text{CTL}}$ equally holds for the $\mathcal{ESG}_{\text{CTL}}^*$ case in light of the fact that the latter subsumes the former.
5.4.4 Eliminating Second-Order Quantifiers

As mentioned above, the definition of the ELIM operation contradicts our assumption that the agent is a pure first-order reasoner. It is however possible to implement ELIM in a way such that the assumption is reestablished, and yet our algorithm remains sound. Recall that we defined ELIM[⟨⟨P_1, ..., P_k⟩⟩, α] to be

$$\exists P_1 \ldots \exists P_n. \alpha.$$ 

If α itself is purely first-order, then replacing a formula of the above form by an equivalent one without any second-order quantifiers is called *predicate quantifier elimination*. The general problem is known to be undecidable [Ack35], yet there a number of techniques as surveyed by Nonnengart et al. [NOS99]. Any method that allows the α to take arbitrary form, such as the well-known SCAN algorithm [GO92], can only be sound, but necessarily has to be incomplete. One approach we can take is to simply plug in any such method as implementation of ELIM. As our overall algorithm is already incomplete, this would “only” introduce a third source of non-termination, in addition to the undecidability of our underlying logic and the possibility for the fixpoint computation loops to not converge.

There are however notable exceptions. Note that in our case, the predicates to be eliminated are the $X_j(\vec{x}_j)$ and $U_i(\vec{x}_i)$ for the temporal subformulas $X\phi_j$ and $\phi_i U \psi_i$ in the input formula, where the arguments $\vec{x}_j$ and $\vec{x}_i$ represent the corresponding subformulas’ free variables. In many practical cases, the auxiliary predicates will be of arity zero, i.e. propositional. This happens whenever quantifiers in the formula do not range into a nested temporal subformula. For example,

$$\langle\langle \delta \rangle \rangle (\forall F(x) \ U (G \exists y F(y)))$$

contains first-order quantification, but the temporal subformulas have no free variables, hence all $U_i$ will be propositional. Eliminating a propositional auxiliary predicate $P$ from α then is nothing more than forgetting [LR94a] the fact $P$ in α. If $\alpha^P_\top$ and $\alpha^P_\bot$ denote α with all free occurrences of $P$ replaced by $\top$ and $\bot$, respectively, we have that

$$\models \exists P \alpha \equiv \alpha^P_\top \lor \alpha^P_\bot.$$ 

The elimination of $P$ thus reduces to a simple syntactical manipulation of formulas, and can easily be iterated for multiple predicates.

There is an even larger set of formulas that we can handle in this manner that even allows for temporal subformulas with free variables. The only constraint we need is that the free variables
in question are quantified outside of the path quantifier, as in:

$$\forall x \langle \delta \rangle GFF(x)$$

The idea is then to use the principle of Universal Generalization that we discussed in Section 3.2.3. Let \( n_1, \ldots, n_k \) be all standard names of the same sort as \( x \) that appear in the basic action theory \( \Sigma \), the program \( \delta \) and the temporal formula \( \varphi \), and let \( n' \) be some standard name that does not occur. Then we can show that

$$\models \Sigma \supset \forall x[\delta] \varphi,$$

if and only if

$$\models \Sigma \supset [\delta]\varphi^n_x$$

for every \( n \in \{ n_1, \ldots, n_k, n' \} \), similar to Theorem 3.15. If \( \varphi \) contains no more quantifiers (or only “safe” ones as in (5.122) above), no temporal subformula in \( \varphi^n_x \) will contain free variables, and we can again use (5.123) to eliminate them. Furthermore, we can treat multiple variables recursively, similar to the RES operator of Definition 3.51.

## 5.5 Terminating Programs

Finally, we have so far only studied the verification of Golog programs that are nonterminating. As argued before, this is justified by the fact that the control program of a robot with an open-ended task necessarily has to be of this form. Nonetheless, there are certain scenarios where we have to deal with terminating programs. For example, we may want to analyze the behaviour of a subroutine of the robot’s overall control program. In this case it is particularly interesting to verify that the procedure ensures that a certain postcondition comes to hold after its execution. In this section, we will therefore briefly address the problem of verifying postconditions for terminating Golog programs.

### 5.5.1 The Algorithm

Here is the algorithm to compute the necessary and sufficient conditions that an execution of program \( \delta \) exists such that the postcondition \( \alpha \) comes to hold after successful termination. Below, let

(5.124) \[
\text{FINAL}[(V, E, v_0), \alpha] \overset{def}{=} \{(v, \alpha \land \varphi) \mid v = \langle \cdot, \varphi \rangle \in V \}
\]
and everything else as in Section 5.3.

**Procedure 7** \( \text{CheckPost}[\delta, \alpha] \)

1. \( L' := \text{Label}[G_\delta, \top]; \)
2. \( L := \text{Final}[G_\delta, \alpha]; \)
3. \( \textbf{while} \ L \neq L' \ \textbf{do} \)
4. \( L' := L; \)
5. \( L := L \lor \text{Pre}[G_\delta, L]; \)
6. \( \textbf{end while} \)
7. \( \textbf{return} \ \text{InitLabel}[G_\delta, L] \)

Intuitively, the procedure starts with labelling each node with the postcondition \( \alpha \), conjoined with the termination condition of that node. We then determine the disjunction of that label set with its preimage, yielding a representation of the configurations where either now or within one step, \( \alpha \) holds and program execution may terminate. When iterating this a second time, we obtain a set of labels that tell us under which conditions we can reach such a configuration in at most two steps, and so on. If the procedure eventually converges, we have found a solution that represents all necessary and sufficient conditions under which there exists a finite run where \( \alpha \) comes to hold after any number of transitions.

Note the close resemblance of the above algorithm with Procedure 4: Again we do an iterative approximation of a least fixpoint, disjoining the current label set with its predecessor in each step. This similarity does not come as a surprise since a successful finite run consists of valid transitions until a final configuration is reached that satisfies the desired postcondition.

**Example 5.63.** As an example, let \( \Sigma \) be as before and \( \delta \) the program

\[
(\pi x. \ Poss(requestCoffee(x))?; requestCoffee(x))^*,
\]

which is basically a simplified, terminating version of our previous \( \delta_{\text{exo}} \). The simple corresponding characteristic graph is depicted in Figure 5.4. Suppose we want to verify whether it is possible to reach a situation where the queue of coffee requests is full after successfully executing this program, i.e. whether

\[
\Sigma \models (\delta)\text{Full(queue)},
\]

where we again assume a queue of size \( k = 2 \). We begin with the initial set of labels

\[
L_0 = \{\langle v_0, \text{Full(queue)} \rangle\}
\]

\[
= \{\langle v_0, \exists x_1 \exists x_2. x_1 \neq e \land x_2 \neq e \land \text{queue} = \langle x_1, x_2 \rangle \rangle\}
\]
\[ \pi x : rC(x)/\text{Poss}(rC(x)) \]

and, since it is not equivalent to \( \top \), determine its preimage:

\[ \text{Pre}[v_0, \mathcal{L}_0] = \{ (v_0, \text{Full}(queue) \lor \text{HalfFilled}(queue)) \} \]

Because our queue has only two slots, the above formula expresses that the queue is half filled. Let therefore \( \text{HalfFilled}(queue) \) denote \( \exists x_1. x_1 \neq e \land \text{queue} = (x_1, e) \). After one iteration, we thus obtain the label set

\[ L_1 = \mathcal{L}_0 \lor \text{Pre}[G_\delta, \mathcal{L}_0] = \{ (v_0, \text{Full}(queue) \lor \text{HalfFilled}(queue)) \} \]

Obviously, \( L_1 \) is not equivalent to \( \mathcal{L}_0 \), therefore another iteration is necessary:

\[ \text{Pre}[v_0, \mathcal{L}_0] = \{ (v_0, \exists x_1 \exists x_2. x_1 \neq e \land \text{queue} = (x_1, x_2)) \} \]

The label set after the second iteration hence is

\[ L_2 = L_1 \lor \text{Pre}[G_\delta, L_1] = \{ (v_0, \text{Full}(queue) \lor \text{HalfFilled}(queue) \lor \text{Empty}(queue)) \} \]
which is again not equivalent to the previous one, so we continue:

\[ \text{Pre}[v_0, L_1] \]
\[ \equiv R[\exists x. \text{Poss}(\text{requestCoffee}(x)) \land \]
\[ \quad [\text{requestCoffee}(x)](\text{Full}(\text{queue}) \lor \text{HalfFilled}(\text{queue}) \lor \text{Empty}(\text{queue})))] \]
\[ \equiv \exists x. x \neq e \land \exists x'. \text{queue} = (x', e) \land \]
\[ \quad \exists x_1 \exists x_2. \text{Enqueue}(\text{queue}, x, (x_1, x_2)) \land (x_1 = e \lor x_2 = e) \]
\[ \equiv \text{queue} = (e, e) \]
\[ \equiv \text{Empty}(\text{queue}) \]

Finally, the resulting set of labels is thus

\[ L_3 = L_2 \text{ OR } \text{Pre}[G_\delta, L_2] \]
\[ = \{ (v_0, \text{Full}(\text{queue}) \lor \text{HalfFilled}(\text{queue}) \lor \text{Empty}(\text{queue})) \} \]
\[ \equiv \{ (v_0, \exists x_1 \exists x_2. \text{queue} = (x_1, x_2) \land (x_1 = e \lor x_2 = e)) \} \]

which is equivalent to \( L_2 \), hence we are done. The resulting formula

\[ \text{Full}(\text{queue}) \lor \text{HalfFilled}(\text{queue}) \lor \text{Empty}(\text{queue}) \]

intuitively describes correctly all and only situations under which we can reach a full queue of requests by means of a sequence of \( \text{requestCoffee}(x) \) actions: If it is full, we are immediately done. When only one of the slots is occupied, we need one \( \text{requestCoffee}(x) \), and in case the queue is empty, two of them are necessary. The equivalent expression

\[ \exists x_1 \exists x_2. \text{queue} = (x_1, x_2) \land (x_1 = e \lor x_2 = e) \]

demonstrates that this means that we only have to require that \( \text{queue} \) is indeed a list of length two and is in some admissible state: Although syntactically nothing can keep us from putting formulas like \( (\text{queue} = \text{bob}) \) or \( (\text{queue} = \langle e, \text{bob} \rangle) \) in \( \Sigma_0 \), the algorithm correctly tells us that we will thus never be able to reach a situation with a full queue.

### 5.5.2 Correctness

Once more we have a theorem that guarantees us the soundness of the procedure:

**Theorem 5.64.** Let \( \Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \) be a basic action theory over \( \langle D, F \rangle \), \( \delta \) a program that only mentions fluents in \( \langle D, F \rangle \) and \( \alpha \) a fluent formula wrt \( \langle D, F \rangle \). Then

\[ \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \models \square \text{CheckPost}[\delta, \alpha] \equiv \langle \delta \rangle \alpha \]
Proof. Let $\mathcal{G}_\delta = (V, E, v_0)$. We prove the equivalent claim that for any world $w$ with $w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$, any $z \in Z$ and any variable map $\theta$, 

$$w, z \models \text{CHECKPost}[\delta, \alpha] \theta \iff w, z \models (\delta \theta) \alpha \theta$$

First consider the simple case where $\text{Final}[\mathcal{G}_\delta, \alpha] \equiv \text{Label}[\mathcal{G}_\delta, \top]$. The procedure then does not enter the while loop, but immediately returns $\alpha \land \varphi_0$ as output, where $\varphi_0$ is the termination condition of the initial node, i.e. $v_0 = (\delta, \varphi_0)$. By assumption, $\alpha \land \varphi_0$ is equivalent to $\top$, which implies that both $\alpha$ and $\varphi_0$ are valid. Then clearly $w, z \models \text{CHECKPost}[\delta, \alpha] \theta$. Moreover, since $w, z \models \varphi_0 \theta$, we get that $\langle v_0, \theta, w, z \rangle \in F_{\mathcal{G}_\delta}$ by Definition 5.15, therefore $\langle z, \delta \theta \rangle \in F^w$ by Lemma 5.19, hence $\langle \rangle \in |\delta \theta|_w$. Together with $w, z \models \alpha \theta$ it follows then that $w, z \models (\delta \theta) \alpha \theta$.

Now assume that there is at least one iteration of the loop. The procedure iteratively computes label sets $L_0, L_1, L_2, \ldots$ such that

(5.125) \hspace{1cm} $L_0 = \text{Final}[\mathcal{G}_\delta, \alpha]$

(5.126) \hspace{1cm} $L_{i+1} = L_i \text{ Or Pre}[\mathcal{G}_\delta, L_i]$

Semantically, this means that

(5.127) \hspace{1cm} $||L_0||_{\Sigma} = ||\text{Final}[\mathcal{G}_\delta, \alpha]||_{\Sigma}$

(5.128) \hspace{1cm} $||L_{i+1}||_{\Sigma} = ||L_i||_{\Sigma} \cup ||\text{Pre}[\mathcal{G}_\delta, L_i]||_{\Sigma}$

By assumption, the procedure terminates, hence there is some smallest index $n$ such that $L_{n+1} \equiv L_n$. Obviously, the output will be a fluent formula. The proof of the main claim of this theorem relies on the following property:

(5.129) \hspace{1cm} $\langle v, \theta, w, z \rangle \in ||L_n||_{\Sigma} \iff w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ and $\langle v, \theta, w, z \rangle \xrightarrow{\mathcal{G}_\delta} (v', \theta', w, z \cdot z')$

such that $\langle v', \theta', w, z \cdot z' \rangle \in F_{\mathcal{G}_\delta}$ and $w, z \cdot z' \models \alpha \theta$

Then we have:

$w, z \models \text{CHECKPost}[\delta, \alpha] \theta$

iff $w, z \models \text{InitLabel}[\mathcal{G}_\delta, L_{n+1}] \theta$ \hspace{1cm} (by assumption)

iff $\langle v_0, \psi_0 \rangle \in L_{n+1}$ and $w, z \models \psi_0 \theta$ \hspace{1cm} (by (5.46))

iff $\langle v_0, \theta, w, z \rangle \in ||L_{n+1}||_{\Sigma}$ \hspace{1cm} (by (5.41))

iff $\langle v_0, \theta, w, z \rangle \in ||L_n||_{\Sigma}$ \hspace{1cm} (since $L_{n+1} \equiv L_n$)

iff $\langle v, \theta, w, z \rangle \xrightarrow{\mathcal{G}_\delta} (v', \theta', w, z \cdot z')$

such that $\langle v', \theta', w, z \cdot z' \rangle \in F_{\mathcal{G}_\delta}$ and $w, z \cdot z' \models \alpha \theta$ \hspace{1cm} (by (5.129))
iff \( z' \in \|G_\delta\|_w(\theta) \) and \( w \cdot z' \models \alpha\theta \)  
(by Definition 5.15)

iff \( z' \in \|\delta\|_w \) and \( w \cdot z' \models \alpha\theta \)  
(by Theorem 5.21)

iff \( w , z \models (\delta\theta)\alpha\theta \)  
(by the semantics)

To prove (5.129), we introduce the following notation for the fact that there is some finite run
starting in \( \langle v, \theta, w, z \rangle \) where \( \alpha\theta \) holds after exactly \( j \) transitions:

(5.130)
\[
\langle v, \theta, w, z \rangle \in \mathcal{P}(j) \overset{\text{def}}{=} w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ and,} \\
\langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v_1, \theta_1, w, z \cdot p_1 \rangle \xrightarrow{G_\delta} \ldots \xrightarrow{G_\delta} \langle v_j, \theta_j, w, z \cdot p_1 \cdots p_j \rangle \\
\text{such that } \langle v_j, \theta_j, w, z \cdot p_1 \cdots p_j \rangle \in \mathcal{F}_{G_\delta} \text{ and } w \cdot z \cdot p_1 \cdots p_j \models \alpha\theta
\]

(5.129) is then equivalent to writing that \( \langle v, \theta, w, z \rangle \in \|L_n\|_\Sigma \) iff for some \( j \geq 0, \langle v, \theta, w, z \rangle \in \mathcal{P}(j) \). First we prove the following weaker property for all \( i \geq 0 \):

(5.131)
\[
\langle v, \theta, w, z \rangle \in \|L_i\|_\Sigma \text{ iff for some } j \text{ with } 0 \leq j \leq i, \langle v, \theta, w, z \rangle \in \mathcal{P}(j)
\]

We will also use that

(5.132)
\[
\text{if } \langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v', \theta'[\bar{x}/\bar{n}], w, z \cdot p \rangle, \text{ then } \alpha\theta[\bar{x}/\bar{n}] = \alpha\theta,
\]

which follows from our distinctiveness assumption for variables: Since the \( \bar{x} \) are \( \pi \)-quantified in \( \delta \), they cannot appear freely in \( \alpha \).

The proof of (5.131) is by induction on \( i \):

- \( i = 0 \):

\[
\langle v, \theta, w, z \rangle \in \|L_0\|_\Sigma \\
\text{iff } \langle v, \theta, w, z \rangle \in \|\text{FINAL}[G_\delta, \alpha]\|_\Sigma \text{ (by (5.127))} \\
\text{iff } v = (\cdot, \varphi') \in V, w, z \models (\varphi' \land \alpha)\theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ (by (5.124))} \\
\text{iff } \langle v, \theta, w, z \rangle \in \mathcal{F}_{G_\delta}, w, z \models \alpha\theta \text{ and } w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}} \text{ (by Definition 5.15)}
\]

- \( i \mapsto i + 1 \):

\[
\langle v, \theta, w, z \rangle \in \|L_{i+1}\|_\Sigma \\
\text{iff } \langle v, \theta, w, z \rangle \in \|L_i\|_\Sigma \text{ or } \langle v, \theta, w, z \rangle \in \|\text{PREDI}[G_\delta, L_i]\|_\Sigma \text{ (by (5.128))} \\
\text{iff } \langle v, \theta, w, z \rangle \in \|L_i\|_\Sigma \text{ or } \\
\langle v, \theta, w, z \rangle \xrightarrow{G_\delta} \langle v', \theta', w, z \cdot p \rangle \text{ and } \langle v', \theta, w, z \cdot p \rangle \in \|L_i\|_\Sigma \text{ (by Lemma 5.32)}
\]
iff for some $j$ with $0 \leq j \leq i$, $(v, \theta, w, z) \in P(j)$ or $\langle v, \theta, w, z \rangle \xrightarrow{\delta} \langle v', \theta', w, z \cdot p \rangle$

and for some $j$ with $0 \leq j \leq i$, $\langle v', \theta', w, z \cdot p \rangle \in P(j)$ (by induction)

iff for some $j$ with $0 \leq j \leq i$, $(v, \theta, w, z) \in P(j)$ or

for some $j$ with $1 \leq j \leq i + 1$, $(v, \theta, w, z) \in P(j)$ (by (5.130) and (5.132))

iff for some $j$ with $0 \leq j \leq i + 1$, $(v, \theta, w, z) \in P(j)$

Finally, we obtain (5.129) from (5.131) as follows. Assume that $n$ is the smallest index such that $L_{n+1} \equiv L_n$, but that the algorithm would apply (5.126) indefinitely. Then $(v, \theta, w, z) \in P(j)$ iff $j \leq n$ and $(v, \theta, w, z) \in \|L_n\|_\Sigma$ by (5.131), or $j > n$ and $(v, \theta, w, z) \in \|L_j\|_\Sigma$ by (5.131), but where $\|L_j\|_\Sigma = \|L_n\|_\Sigma$ due to the fact that $\|L_m\|_\Sigma = \|L_n\|_\Sigma$ for $m \geq n$, which follows inductively from $L_{n+1} \equiv L_n$.

As a final remark, note that this new procedure could be used to extend the regression operator such that it can handle formulas that not only contain $[t]$ operators with primitive actions, but also $[\delta]$ containing entire programs. We would then simply add the rule

12. $R[\sigma, \langle \delta \rangle \alpha] = R[\sigma, \text{CheckPost}[\delta, R[\alpha]]]$.

Although again we then cannot guarantee termination anymore, this would even allow us to handle the most interesting case of programs containing iteration. Moreover, for any program that does not contain iteration, it is easy to see that the procedure will terminate due to the fact that the corresponding characteristic graph does not contain any cycles.

### 5.6 BDD Representation

Before concluding this chapter, we want to discuss some suggestions for the possible implementation of the verification procedures that were presented in this thesis. As we have seen, the algorithms heavily rely on regression. The latter tends to quickly “blow up” formulas: Typically, the size of the regression result is roughly exponential in the number of nested action operators in the original formula, or in our case roughly exponential in the number of iterations of the fixpoint computation loop. Moreover, when the definitional and successor state axioms of the basic action theory contain quantification, each regression step introduces new quantifiers.

As is to be expected, theorem-proving systems perform worse the larger the input formula and the more quantifiers it contains. A naive implementation of our algorithm that does syntactical manipulations of formulas strictly according to the theorems presented in this chapter
and that then calls a theorem prover to decide the equivalence of label formulas will show a very bad performance, even for small problems. This is even true in spite of the fact that theorem provers often do a certain preprocessing to simplify input formulas and bring them into their internal representational form. The reason is that the effort for this preprocessing has to be spent anew in each single iteration, and that it has to deal with formulas of increasing size.

In a more sophisticated implementation of the algorithms we would therefore rather want to keep formulas as compact as possible and maintain this form throughout the algorithm’s entire operation. Standard clausal-form representations such as CNF and DNF have the disadvantage that the former tends to be compact only for formulas with many conjunctions and few disjunctions, and vice versa for the latter. Unfortunately, the formulas produced by our algorithms make heavy use of both connectives.

When it comes to propositional formulas, ordered binary decision diagrams (OBDDs) [Bry86] have become an increasingly popular and efficient means of representation, in particular as the basis for symbolic model checking [BCM+92]. They are somewhat similar to binary decision trees, except that they are actually directed acyclic graphs and that an order over the propositional variables is assumed. Figure 5.5 shows an example. An OBDD is often much more compact than representations in other forms. They also support an efficient manipulation through Boolean operations. What is particularly interesting about them is that if the variable ordering is fixed, the fully reduced form of an OBDD is unique. Typically, implementations materialize the OBDD of each subformula only once in memory and represent each formula by a pointer to the node that is the root of the corresponding OBDD. It is thus possible to check the equivalence of two formulas in constant time, as they are equivalent just in case their pointers refer to the same root node!

A variety of approaches have been proposed for lifting OBDDs to the first-order case. While
many of them require the formula to be in prenex form or quantifier-free \cite{GT03, WJK07}, a variant of the first-order algebraic decision diagrams (FOADDs) introduced by Sanner \cite{San08} seems best suited for our purposes. The idea is to “expose the propositional structure of a first-order formula” by pushing quantifiers inside as deeply as possible. For this purpose, Sanner suggests to repeatedly apply the following rewrite rules:

\begin{align*}
\exists x. \phi(x, \cdot) \lor \psi(x, \cdot) & \rightarrow (\exists x \phi(x, \cdot)) \lor (\exists x \psi(x, \cdot)) \tag{5.133} \\
\forall x. \phi(x, \cdot) \land \psi(x, \cdot) & \rightarrow (\forall x \phi(x, \cdot)) \land (\forall x \psi(x, \cdot)) \tag{5.134} \\
\exists x. \phi(x, \cdot) \land \psi(\cdot) & \rightarrow (\exists x \phi(x, \cdot)) \land \psi(\cdot) \tag{5.135} \\
\forall x. \phi(x, \cdot) \lor \psi(\cdot) & \rightarrow (\forall x \phi(x, \cdot)) \lor \psi(\cdot) \tag{5.136}
\end{align*}

Furthermore, quantifiers can be eliminated from the input formula using:

\begin{align*}
\exists x. x = y \land \phi(x, \cdot) & \rightarrow \phi(y, \cdot) \tag{5.137} \\
\forall x. x \neq y \lor \phi(x, \cdot) & \rightarrow \phi(y, \cdot) \tag{5.138}
\end{align*}

For example, the formula

\[ \exists x. (P(x) \lor \forall y. P(x) \land Q(x) \land \neg P(y)) \]

is rewritten to

\[ \exists x P(x) \lor \left( \exists x (P(x) \land Q(x)) \land \forall y \neg P(y) \right). \]

The boxes illustrate the propositional structure of the formula. In its OBDD representation, each box is treated as a propositional atom. If \( A \) stands for \( \exists x P(x) \) and \( B \) for \( \exists x P(x) \land Q(x) \), then we obtain exactly the OBDD depicted in Figure 5.5.

Obviously, the property that equivalent formulas have the same OBDD seizes to hold when we move to the first-order case. Otherwise, we could thus decide first-order validity by simply checking whether the OBDD of a given sentence is identical to the OBDD only consisting of the leaf node 1. The reason for this lies in the fact that the construction procedure for OBDDs sketched above may fail to detect that quantified subformulas that it regards as distinct are indeed equivalent. Consequently, checking equivalence of first-order formulas by comparing their OBDDs is only sound, but incomplete.

We may be willing to accept that in light of the fact our algorithms are incomplete anyhow. More importantly, observe that this procedure is sufficient in many practical cases. For instance, take the labels of Example 5.63. When we simplified formulas, it turned out that they were always Boolean combinations of a finite set of quantified subformulas for which we introduced the
abbreviations $\text{Empty}(\text{queue})$, $\text{HalfFilled}(\text{queue})$ and $\text{Full}(\text{queue})$. This is not surprising given that a queue with only two slots will necessarily always satisfy exactly one of these three cases, which can also be viewed as abstract states. What makes the usage of OBDDs as described above especially appealing is that their construction would automatically expose this matter of fact, yielding a very compact representation in which each of the three subformulas corresponds to one propositional atom.

We did some preliminary experimentation with a prototypical implementation, which showed encouraging first results. An in-depth evaluation and formal analysis is a subject for future work.

## 5.7 Discussion

### 5.7.1 Summary

This chapter presented the verification of non-terminating (and also terminating) GOLOG programs. We started with motivating that it is equally essential to verify that a GOLOG agent meets its specification and requirements before deployment as it is for any other hardware and software product. It was argued that while either proving properties manually or resorting to existing model checking methods would constitute viable alternatives, a holistic approach would be more desirable where an automated verification is done in the very same expressive formalism and with the very same reasoning mechanisms that are used for the formalization and actual control of the agent. Therefore, we proposed an extension of the logic $\mathcal{ES}$ by two new modal operators that allow to express properties of execution paths of GOLOG programs, drawing inspirations from temporal logics [Eme90], process logic [HKP82], and dynamic logic [HKT00]. Next, we introduced characteristic graphs as a means of representing the possible execution traces of a GOLOG program. We then presented automatic verification algorithms for properties of non-terminating programs expressed in a CTL-like subset of the logic, after which we considered a more expressive $\text{CTL}^*$-like subclass, before we addressed verifying postconditions of terminating programs. Finally, we discussed the implementation of these algorithms by means of a first-order variant of binary decision diagrams.

### 5.7.2 Comparison to Related Work

As remarked earlier, the verification of non-terminating GOLOG programs has received surprisingly little attention so far, one notable exception being the work by De Giacomo, Ternovska and Reiter [GTR97] who however only do manual, meta-theoretic proofs within second-order logic. Liu [Liu02] presents a Hoare-style proof system for the partial correctness of terminating
5.7 Discussion

Golog programs. She proves its soundness and, for a subset of programs, also its “relative completeness” [Coo78]. The latter refers to completeness under the assumption that we are given an oracle that decides the truth of first-order formulas, a notion that is closely related to our assumption of the agent being a logically omniscient first-order reasoner.

Kelly and Pearce [KP10] study property persistence in the classical Situation Calculus using a technique very similar to ours that is based on an iterative fixpoint approximation. However, they do not restrict the space of reachable situations by Golog programs. In fact, property persistence can be viewed as a special case of what our algorithm for \( \mathcal{ES}_{\text{CTL}} \) can handle, considering that the formula \( \square \alpha \) (i.e. \( \alpha \) persists to hold) is equivalent to \([\text{any}^{\omega}]G \alpha \) in \( \mathcal{ES} \).

The earlier mentioned work of Baader, Liu and ul Mehdi [BLuM10] is one option to obtain a decidable verification procedure for properties of action logic programs. First, instead of using the full first-order expressiveness of the Situation Calculus or \( \mathcal{ES} \), they resort to a dynamic extension [BLM+05] of the decidable description logic \( \mathcal{ALC} \) [BCM+03] to represent pre- and postconditions of actions, where properties are expressed by a variant of LTL over \( \mathcal{ALC} \) propositions [BGL08]. Second, they encode programs by finite Büchi automata. Thus they indeed handle a proper subset of possible inputs for our algorithm since their \( \mathcal{ALC} \)-based formalism can be viewed as a fragment of the Situation Calculus, and since we can simulate the state transitions of a finite Büchi automaton through a BAT and express its acceptance criterion by means of a temporal formula.

Finally, the idea to verify Golog programs by means of iterative fixpoint computations using characteristic program graphs has recently been taken on by De Giacomo, Lespérance and Pearce [GLP10] in the context of games and multi-agent systems, where properties are expressed in Alternating-Time Temporal Logic [AHK02].

5.7.3 Future Work

There still remain many possible directions for future work:

Decidability: It is not hard to see that the problem of verifying \( \mathcal{ES} \) formulas is highly undecidable. As a consequence, any algorithm that attempts to solve it, such as the ones presented in this chapter, can at most guarantee soundness, but not termination. It should be emphasized again that was presented here were merely idealized reasoning methods, similar in spirit to Reiter’s regression or Lakemeyer and Levesque’s Representation Theorem in the sense that they show how a query can be reduced to classical FOL theorem proving by iteratively “factoring out” its non-classical aspects.

One could take the stance that this is completely sufficient, and that it lies in the domain
designer’s responsibility to devise a basic action theory and a program for the agent that ensure that reasoning remains feasible. Our algorithms could then serve as a first sanity check in the sense that if we get “stuck” in deciding first-order entailments during the offline verification of a program, then this is an indication of a bad design as it is very likely that this also happens at runtime after deploying the program.

Nevertheless it is probably more desirable to have guaranteed termination. As discussed before, this is under the premises that we are able to use the same formalism both for the specification and the control of the agent, and that we retain as much expressiveness of first-order logic as possible to cope with possibly unknown individuals. Baader, Liu and ul Mehdi’s aforesaid method based on description logics and Büchi automata is certainly one first step in this direction deserving further attention. The same goes for applying Liu, Lakemeyer and Levesque’s incomplete, yet tractable reasoning method [LLL04, LL05b] in combination with tractable progression of proper+ knowledge bases for local-effect action theories [LL09b], as mentioned in Section 5.3.3. Finally, it would be interesting to expand upon results of a recent diploma thesis [Lie13] that explores how using an CS variant of Gu and Soutchanski’s decidable two-variable Situation Calculus [GS10] together with a restriction to well-known classes of SSAs such as the context-free ones [LR97] or those with only local effects [LL05b] influences the termination behaviour of our verification algorithms.

**Witnesses and Counterexamples:** When verifying systems using model checking, it is often useful if the system can provide information beyond a simple “yes” or “no” answer. If we have a query that asks whether a path with a certain property exists, then a *witness* is given in the positive case. Similarly, if a formula is supposed to hold for all execution paths and it turns out that it does not, then a *counterexample* can give valuable insight into the potential error source.

In a sense, our algorithms already do provide witnesses and counterexamples, namely in the form of the fluent formulas that are returned by the corresponding procedures. If for instance a call to CHECKEG[δ, ϕ] yields ϕ as output, then it means that an execution trace of δ satisfying Gϕ exists just in case ϕ holds initially, and that the path in question must be some trace admitted by ϕ?; δ. There are nonetheless scenarios where the latter is not sufficient to uniquely identify a troublesome execution of a program. A possible direction for future work therefore is to extend the verification algorithms by a mechanism that constructs witnesses and counterexamples, for instance in the form of deterministic GOLOG programs.
Semantics: Although $\mathcal{ES}$ is sufficiently expressive for representing and verifying program properties in many application scenarios, it nonetheless suffers from a few weaknesses. First, the semantics of programs as given by Definitions 5.5 to 5.7 was devised to resemble the semantics of $\omega$-regular expressions. In particular, one intention behind it was that the $\cdot^\omega$ operator only admits infinite traces, while the $\cdot^*$ only admits finite ones. However, the rather unusual requirement that an infinite trace is one that never visits a final configuration sometimes leads to unintuitive results. For example, it is not possible to express a “mixed” program admitting both finite and infinite traces. The reader may verify that the program $a^* | b^\omega$ does not yield any infinite runs because all reachable configurations are final. Moreover, in a program such as $a^*; b^\omega$, the supposedly finite iteration $a^*$ may indeed be executed infinitely since executing $a$ once leads back to a configuration with $a^*; b^\omega$ as remaining program, which is obviously not final. It would be desirable to come up with an alternative program semantics that does not suffer from these shortcomings.

Another intention behind $\mathcal{ES}$ was to generalize the branching time temporal logics CTL and CTL$^*$. In particular, the existential and universal path quantifiers $E$ and $A$ may be viewed as special cases of the $\mathcal{ES}$ program quantifiers $\langle \delta \rangle$ and $[\delta]$, respectively. The correspondence is, however, not quite exact, namely when it comes to the nesting of such quantifiers. In classical branching time temporal logics, one can use a formula such as $AGEF\text{Occ}(\text{recharge})$ to express that on all paths it is always possible to eventually reach a certain desired state. With the current semantics, we cannot state the same in $\mathcal{ES}$. The reason is that the nested $\langle \cdot \rangle$ quantifier in the formula $[\delta]G\langle \text{any}^\omega \rangle F\text{Occ}(\text{recharge})$ is completely independent from the outer $[\delta]$. Instead, we may rather want the inner $\langle \text{any}^\omega \rangle$ to be restricted to (continuations of) the paths admitted by the encapsulating quantifier. While the former reading was incorporated in the definition of $\mathcal{ES}$ because it yields a much simpler semantics, a suitable adoption of the latter would increase the expressiveness of the language.

Epistemic Operators: In this chapter we only verified objective formulas. It would be interesting to see whether we can lift our verification methods to the case of an agent with incomplete information that is controlled by a knowledge-based program as described in Section 3.7. At first glance, it seems enough to simply apply the methods presented in Section 3.5 to reduce reasoning about action and knowledge to first-order theorem proving. However, the elimination of epistemic modalities in the Representation Theorem happens always with respect to a given static knowledge base, which in our current formulation is only available for the initial situation. We would therefore have to revise our algorithms such that instead of iteratively determining preimages by regressing label formulas, they perform a forward exploration of the
state space by means of repeatedly *progressing* the agent’s initial theory. Alternatively, it might be interesting to see whether the elimination of knowledge operators can be avoided altogether, for example by resorting to incomplete inference methods for subjective formulas similar to the ones employed in Petrick and Bacchus’ knowledge-based planner PKS [PB02, PB04].

**Larger Subsets of Golog:** The restricted subset of GOLOG that was used in this chapter misses some of the language’s interesting constructs, in particular procedures. The problem is that when we do transitions in a program containing (recursive) procedure calls, the reachable subprograms of intermediate configurations include the current procedure call stack. Since those stacks may become arbitrarily large, the space of reachable subprograms is not finite anymore, and we cannot encode the program through a characteristic graph. The difficulty in extending our verification algorithms to a more general subclass of GOLOG programs therefore lies in appropriately representing the corresponding infinite configuration space.
Chapter 6

Conclusion

The knowledge-based approach to agent design is particularly suited in our envisioned scenario of an autonomous, domestic robot performing an open-ended task. The Situation Calculus, an expressive dialect of classical first-order logic, is probably the best known and most widely studied action logic for modelling such dynamic domains. It also forms the basis for GOLOG, which is used to define complex behaviours for agents and allows to freely combine programming with planning. Among other things, GOLOG has been successfully applied to the control of mobile robots.

This thesis identified and tackled three issues with agents defined by means of the classical Situation Calculus and GOLOG. The first one occurs in the realistic case of an agent with incomplete world knowledge that needs to use sensing to gather information at runtime, where epistemic modalities are included in the language to express statements about what the agent knows and does not know. The corresponding Situation Calculus background axiomatization of a possible-world semantics is quite involved and sometimes leads to counter-intuitive results. Lakemeyer and Levesque’s logic $\mathcal{ES}$ is a modal variant of the epistemic Situation Calculus where belief and action is encoded through modalities that are defined within the (non-classical) semantics of the logic, rather than axiomatically. This yields a better readable syntax and much easier proofs, while all the benefits of the Situation Calculus are retained, including Reiter’s solution to the frame problem, regression- and progression-based reasoning, first-order expressiveness, and GOLOG. In this thesis, a formal account of knowledge-based agents on the basis of $\mathcal{ES}$ was developed. This includes the integration of various reasoning procedures to handle actions, sensing and knowledge, an $\mathcal{ES}$-based transition semantics for a rich variant of GOLOG, as well as a formal interface definition for the meta-level control that implements the sense-plan-act cycle of the agent.

Second, in many scenarios agents often encounter subproblems that are rather combinatorial
in nature, such as scheduling pending requests, planning a route, or a combination of the two. Although GOLOG supports planning in principle, existing implementations typically perform very bad on such pure planning tasks, especially in comparison to state-of-the-art planners such as those participating at the biennial International Planning Competition. We therefore addressed the integration of such planners into GOLOG, with the aim of obtaining a system that combines the expressiveness of GOLOG with the efficiency of planning. As theoretical foundation, we developed a mapping from an important fragment of the planning domain definition language PDDL, the de-facto standard for the formulation of planning problems, to $\mathcal{ES}$, thus obtaining an alternative, declarative semantics for PDDL that is in our view easier to grasp than the standard meta-theoretic definition. An empirical evaluation was conducted that showed that embedding state-of-the-art planners is indeed highly advantageous in terms of an increase of the overall runtime performance of the system, while none of GOLOG’s expressiveness has to be given up.

Third, before deploying a GOLOG program to the robot and executing it physically, it is often desirable if not crucial to verify that it indeed fulfills its intended purpose and meets certain requirements. In the case of an autonomous agent with an open-ended task, the control program is usually a non-terminating one. Surprisingly, the verification of non-terminating GOLOG programs had previously received little attention, except for work that discusses manual, meta-theoretic proofs of properties expressed in terms of inductive fixpoint definitions. As this is prone to errors and difficult to grasp even for the mathematically inclined, an automated verification is much more desirable. For this purpose, we proposed an extension to $\mathcal{ES}$ that includes new modal operators to express temporal properties of GOLOG programs, and provided algorithms for their automated verification. They rely on a newly introduced graph representation for GOLOG programs called characteristic graphs which are used to systematically explore the state space. Similar to other forms of reasoning in the Situation Calculus, our verification methods ultimately reduce to classical first-order theorem proving, and therefore only their soundness can be guaranteed.

In addition to the possible directions for future work that were discussed in the individual chapters, one interesting possibility for further research lies in combining the issues addressed in this thesis. After all, planning and verification are related forms of reasoning: Planning is the process of finding an action sequence (or more generally, a policy or program) that ensures that a goal property will come to hold. Similarly, verification aims at either establishing or disproving a given property, preferably producing a corresponding witness or counterexample in the form of a path (or policy or program). Planning could therefore be viewed as a special form of verification, and vice versa. In the context of a GOLOG system, this could mean that we refrain
from having separate reasoning procedures for the interpreter’s lookahead on the one hand and verification on the other hand, but devise a single algorithm that is suited and practicable for both purposes by synthesizing a controller or program out of a formal specification.
Appendix A

Proofs

A.1 Proof of Theorem 3.39

Theorem 3.39 Let $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ be a basic action theory over fluents $(D, F)$. For every relational $F_i \in F$, we introduce a new predicate symbol $P_i$ with the same arity. Similarly, for every functional $f_i \in F$, let $h_i$ be a new function symbol of the same arity and sort. For any formula $\phi$, let $\phi^{\vec{F}, \vec{f}}$ denote the result of replacing every occurrence of $F_i$ by $P_i$ and every occurrence of $f_i$ by $h_i$. Then the following is a progression of $\Sigma_0$ through $t$ wrt $\Sigma_{\text{post}}$:

\[(A.1) \quad \exists P_1 \ldots \exists P_l \exists h_1 \ldots \exists h_m. \Psi,\]

where $\Psi$ refers to the conjunction of the following:

\[(A.2) \quad (\Sigma_0)^{\vec{F}, \vec{f}}_{\vec{P}, \vec{h}}\]

\[(A.3) \quad \bigwedge_{i=1}^{l} \forall \vec{x}_i \left( F_i(\vec{x}_i) \equiv (\gamma_{F_i}^{\vec{x}})^{\vec{P}, \vec{h}}_{\vec{P}, \vec{h}} \right)\]

\[(A.4) \quad \bigwedge_{i=1}^{m} \forall \vec{x}_i \forall y_i \left( (f_i(\vec{x}_i) = y_i) \equiv (\gamma_{f_i})^{\vec{x}}^y_{\vec{P}, \vec{h}} \right)\]

Proof. Let $w'$ be a world and let $\Sigma'_0$ denote (A.1).

$\Rightarrow$: Let $w' \models \Sigma'_0$. We have to show that there is some world $w$ such that $w \models \Sigma$ and $w''_{\Sigma} = w_p$, where $p = |t|^w_{\vec{P}, \vec{h}}$. By Definition 3.22, there is a world $w''$ such that $w'' \sim_{\vec{P}, \vec{h}} w'$ and $w'' \models \Psi$. We define $w'''$ to be a world such that for all the $F_i$, all the $f_i$ and all standard names $\vec{n}$,

\[w'''[F_i(\vec{n}), \{\}] = w''[P_i(\vec{n}), \{\}]\]

\[w'''[f_i(\vec{n}), \{\}] = w''[h_i(\vec{n}), \{\}]\]
and for all $G, g \notin D \cup F \cup \{P_1, \ldots, P_l\} \cup \{h_1, \ldots, h_m\}$, all $\vec{n}$, and all $z \in \mathbb{Z}$,
\[
\begin{align*}
  w'''[G(\vec{n}), p \cdot z'] &= w''[G(\vec{n}), z'] \\
  w'''[g(\vec{n}), p \cdot z'] &= w''[g(\vec{n}), z']
\end{align*}
\]
and further that for all $P_i$, all $h_i$, all $\vec{n}$, and all $z' \in \mathbb{Z}$,
\[
\begin{align*}
  w'''[P_i(\vec{n}), p \cdot z'] &= w'[P_i(\vec{n}), z'] \\
  w'''[h_i(\vec{n}), p \cdot z'] &= w'[h_i(\vec{n}), z']
\end{align*}
\]
Now let $w = w'''$. We will need the property that for any fluent sentence $\phi$,
\[(A.5)\quad w'' \models \phi \text{ iff } w''' \models \phi,\]
First we show that for any fluent ground term $t''$,
\[(A.6)\quad |t''^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} = |t''|_{w''},\]
where $t''^{\vec{f}^{\vec{h}}}_{\vec{h}}$ denotes the result of replacing every occurrence of $f_i$ by the corresponding $h_i$ within the term $t''$. The proof is by an induction on the structure of $t''$:

- $t'' = n \in \mathbb{N}$: $|t''^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} = n = |t''|_{w''}$.
- $t'' = f_i(t_1, \ldots, t_k)$, $f_i \in F$:
  \[
  |f_i(t_1, \ldots, t_k)^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} = |f_i(t_1, \ldots, t_k)|_{w''} \quad \text{(by definition of } \vec{f}^{\vec{h}}_{\vec{h}})\]
  \[
  = |h_i(t_1, \ldots, t_k)^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} \quad \text{(by the semantics)}
  = w''[h_i(n_1, \ldots, n_k), \emptyset], \text{ where } n_j = |t_j^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w''} \quad \text{(by assumption)}
  = w''[f_i(n_1, \ldots, n_k), \emptyset], \text{ where } n_j = |t_j^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w''} \quad \text{(by induction)}
  = |f_i(t_1, \ldots, t_k)|_{w''} \quad \text{(by the semantics)}
  \]
- $t'' = g(t_1, \ldots, t_k)$, $g$ rigid:
  \[
  |g(t_1, \ldots, t_k)^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} = |g(t_1, \ldots, t_k)|_{w''} \quad \text{(by definition of } \vec{f}^{\vec{h}}_{\vec{h}})\]
  \[
  = |g(t_1, \ldots, t_k)^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w'''} \quad \text{(by the semantics)}
  = w''[g(n_1, \ldots, n_k), \emptyset], \text{ where } n_j = |t_j^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w''} \quad \text{(by assumption)}
  = w''[g(n_1, \ldots, n_k), p], \text{ where } n_j = |t_j^{\vec{f}^{\vec{h}}}_{\vec{h}}|_{w''} \quad \text{(by assumption)}
  \]
A.1 Proof of Theorem 3.39

\[
= w''[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j^f_{w''}|
\]
(by the rigidity constraint)

\[
= w''[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j^{0}_{w''}|
\]
(by induction)

\[
= |g(t_1, \ldots, t_k)|_{w''}
\]
(by the semantics)

Now we can prove (A.5) by an induction on \(\phi\):

- **\(\phi = (t_1 = t_2)\):**

\[
w'' \models (t_1 = t_2) \iff w'' \models (t_1^f = t_2^f) \iff |t_1^f|_{w''} = |t_2^f|_{w''}
\]
(by definition of \(\frac{\hat{F}}{\hat{P}} \frac{\hat{f}}{\hat{h}}\))

- **\(\phi = F_i(t_1, \ldots, t_k), F_i \in \mathcal{F}\):**

\[
w'' \models F_i(t_1, \ldots, t_k) \iff w'' \models P_i(t_1^f, \ldots, t_k^f) \iff w''[P_i(n_1, \ldots, n_k), \langle \rangle] = 1, \text{ where } n_j = |t_j^f|_{w''}
\]
(by the semantics)

- **\(\phi = G(t_1, \ldots, t_k), G \text{ rigid}\):**

\[
w'' \models G(t_1, \ldots, t_k) \iff w'' \models G(t_1^f, \ldots, t_k^f) \iff w''[G(n_1, \ldots, n_k), \langle \rangle] = 1, \text{ where } n_j = |t_j^f|_{w''}
\]
(by the semantics)
• \( \phi = \phi_1 \land \phi_2 \):

\[
\begin{align*}
w'' &\models (\phi_1 \land \phi_2)_{\vec{P}_{\vec{R}}_{\vec{h}}} \\
\text{iff } w'' &\models \phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}} \land \phi_2_{\vec{P}_{\vec{R}}_{\vec{h}}} & \text{(by definition of } \phi_{\vec{P}_{\vec{R}}_{\vec{h}}}) \\
\text{iff } w'' &\models \phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}} \text{ and } w'' &\models \phi_2_{\vec{P}_{\vec{R}}_{\vec{h}}} & \text{(by the semantics)} \\
\text{iff } w'' &\models \phi_1 \text{ and } w'' &\models \phi_2 & \text{(by induction)} \\
\text{iff } w'' &\models \phi_1 \land \phi_2 & \text{(by the semantics)}
\end{align*}
\]

• \( \phi = \neg \phi_1 \):

\[
\begin{align*}
w'' &\models (\neg \phi_1)_{\vec{P}_{\vec{R}}_{\vec{h}}} \\
\text{iff } w'' &\models \neg (\phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}}) & \text{(by definition of } \phi_{\vec{P}_{\vec{R}}_{\vec{h}}}) \\
\text{iff } w'' &\not\models \phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}} & \text{(by the semantics)} \\
\text{iff } w'' &\not\models \phi_1 & \text{(by induction)} \\
\text{iff } w'' &\models \neg \phi_1 & \text{(by the semantics)}
\end{align*}
\]

• \( \phi = \forall x \phi_1 \):

\[
\begin{align*}
w'' &\models (\forall x \phi_1)_{\vec{P}_{\vec{R}}_{\vec{h}}} \\
\text{iff } w'' &\models \forall x (\phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}}) & \text{(by definition of } \phi_{\vec{P}_{\vec{R}}_{\vec{h}}}) \\
\text{iff } w'' &\models (\phi_1_{\vec{P}_{\vec{R}}_{\vec{h}}})_{\vec{P}_{\vec{R}}_{\vec{h}}} \text{ for all } n \in \mathbb{N}_x & \text{(by the semantics)} \\
\text{iff } w'' &\models (\phi_1^n)_{\vec{P}_{\vec{R}}_{\vec{h}}} \text{ for all } n \in \mathbb{N}_x & \text{(see below)} \\
\text{iff } w'' &\models \phi_1^n \text{ for all } n \in \mathbb{N}_x & \text{(by induction)} \\
\text{iff } w'' &\models \forall x \phi_1 & \text{(by the semantics)}
\end{align*}
\]

Above we made use of the fact that the order of the \( x \) and \( \vec{P}_{\vec{R}} \) substitutions is in fact irrelevant as the former is concerned with the variable \( x \), while the latter only touches the predicate and function symbols \( \vec{F} \) and \( \vec{f} \), i.e. they do not interfere with each other. In general, if \( \phi \) is a formula and \( x \) is a substitution of variables by terms, then

\[
(A.7) \quad (\phi_{\vec{F}_{\vec{P}_{\vec{R}}_{\vec{l}}}})^{\vec{F}_{\vec{P}_{\vec{R}}_{\vec{l}}}} = (\phi_{\vec{P}_{\vec{R}}_{\vec{l}}})_{\vec{F}_{\vec{P}_{\vec{R}}_{\vec{l}}}}^x.
\]

Since \( w'' \models (\Sigma_0)_{\vec{P}_{\vec{R}}_{\vec{h}}} \) by assumption, it now follows from \((A.5)\) that \( w'' \models \Sigma_0 \). By Lemma 3.32 (2) we get \( w''_\Sigma \models \Sigma \).
As for showing that \( w'_\Sigma = w_p \), according to Definition 3.35, this means that for all \( z' \in Z \), all primitive formulas \( \beta \) and all primitive terms \( t' \):

\begin{align}
(A.8) & \quad w'_\Sigma[t', z'] = w[t', p \cdot z'] \\
(A.9) & \quad w'_\Sigma[\beta, z'] = w[\beta, p \cdot z']
\end{align}

To prove this, we show a more general property, namely that for all ground terms \( t'' \) and all static, objective sentences \( \phi \):

\begin{align}
(A.10) & \quad |t''|^{z'}_{w_\Sigma} = |t''|^{p \cdot z'}_w \\
(A.11) & \quad w'_\Sigma, z' \models \phi \iff w''_\Sigma, p \cdot z' \models \phi
\end{align}

Properties (A.8) and (A.9) follow then as special cases from (A.10) and (A.11), respectively, since any primitive term is also a ground term, and any primitive formula is also a static, objective sentence. The proof is on both theses properties together, with an outer induction on the length of the action sequence \( z' \) and a sub-induction on the size of \( t'' \) and \( \phi \), where any occurrence of \( d(\vec{t}) \) is counted as the size of the corresponding \( \varphi_{d'} + 1 \) and any occurrence of \( D(\vec{t}) \) as the size of \( \varphi_{D'} + 1 \). The induction is well-behaved since the formulas \( \varphi_d, \varphi_D, \gamma_{f_i} \) and \( \gamma_{\vec{f}_i} \) are fluent formulas wrt \( \langle D, F \rangle \) and therefore do not mention any further \( d \in D \) or \( D \in D \). Note that below the distinction between \( z' = \langle \rangle \) and \( z' = z'' \cdot p' \) for the outer induction is only necessary in case of \( t'' = f_i(t_1, \ldots, t_k) \) and \( \phi = F_i(t_1, \ldots, t_k) \), since for all the other symbols and constructs, the two cases can be proven in an identical manner.

- \( t'' = n \in N' \): \( |t''|^{p \cdot z'}_w = n = |t''|^{z'}_{w_\Sigma} \)
- \( t'' = f_i(t_1, \ldots, t_k), f_i \in F' \):
  - \( z' = \langle \rangle \):
    - \( |f_i(t_1, \ldots, t_k)|^p_w = n \)
    - \( w[f_i(n_1, \ldots, n_k), p] = n, \) where \( n_j = |t_j|^p_w \) \hspace{1cm} (by the semantics)
    - \( w[f_i(n_1, \ldots, n_k), p] = n, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (by induction)
    - \( w = \gamma_{f_i^{x_1 \cdots x_k y_1 \cdots y_l a}_n}, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (since \( w \models \Sigma_{\text{pos}} \))
    - \( w = \gamma_{f_i^{x_1 \cdots x_k y_1 \cdots y_l a}_n}, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (since \( p = |t|^0_w \) by assumption)
    - \( w'' = \mathcal{R}[\gamma_{f_i^{x_1 \cdots x_k y_1 \cdots y_l a}_n}, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (by Lemma 3.32 (4))
    - \( w'' = \gamma_{f_i^{x_1 \cdots x_k y_1 \cdots y_l a}_n}, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (by Lemma 3.32 (5))
    - \( w'' = (\gamma_{f_i^{x_1 \cdots x_k y_1 \cdots y_l a}_n t} p f_{\vec{h}} f_{\vec{k}}, \) where \( n_j = |t_j|^0_{w_\Sigma} \) \hspace{1cm} (by (A.5))
iff \( w'' \models f_i(n_1, \ldots, n_k) = n \), where \( n_j = |t_j|_{w''}^0 \) (since \( w'' \models (A.4) \))

iff \( w'[f_i(n_1, \ldots, n_k), \emptyset] = n \), where \( n_j = |t_j|_{w''}^0 \) (since \( w'' \sim_{\mathcal{F}, \mathcal{P}} w' \))

iff \( w'_\Sigma[f_i(n_1, \ldots, n_k), \emptyset] = n \), where \( n_j = |t_j|_{w'_\Sigma}^0 \) (by Definition 3.31)

iff \( |f_i(t_1, \ldots, t_k)|_{w'_\Sigma}^{0} = n \) (by the semantics)

- \( z' = z'' \cdot p' \):

\[
|f_i(t_1, \ldots, t_k)|_{w''}^{p''} = n
\]

iff \( w[f_i(n_1, \ldots, n_k), p \cdot z'' \cdot p'] = n \), where \( n_j = |t_j|_{w''}^{p''}p' \) (by the semantics)

iff \( w[f_i(n_1, \ldots, n_k), p \cdot z'' \cdot p'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z''}p' \) (by induction)

iff \( w, p \cdot z'' = \gamma_{j_{n_1 \cdots n_k} n} a \), where \( n_j = |t_j|_{w'_\Sigma}^{z''}p' \) (since \( w \models \Sigma_{\text{post}} \))

iff \( w'_\Sigma, z'' = \gamma_{j_{n_1 \cdots n_k} n} a \), where \( n_j = |t_j|_{w'_\Sigma}^{z''}p' \) (by induction)

iff \( w'_\Sigma[f_i(n_1, \ldots, n_k), z'' \cdot p'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z''}p' \) (since \( w'_\Sigma \models \Sigma_{\text{post}} \))

iff \( |f_i(t_1, \ldots, t_k)|_{w'_\Sigma}^{z''}p' = n \) (by the semantics)

- \( t'' = d(t_1, \ldots, t_k) \), \( d \in \mathcal{D} \):

\[
|d(t_1, \ldots, t_k)|_{w'}^{p''} = n
\]

iff \( w[d(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|_{w'}^{p''} \) (by the semantics)

iff \( w[d(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by induction)

iff \( w, p \cdot z' = \varphi_{d_{n_1 \cdots n_k} n} y \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (since \( w \models \Sigma_{\text{def}} \))

iff \( w'_\Sigma, z' = \varphi_{d_{n_1 \cdots n_k} n} y \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by induction)

iff \( w'_\Sigma[d(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (since \( w'_\Sigma \models \Sigma_{\text{def}} \))

iff \( |d(t_1, \ldots, t_k)|_{w'_\Sigma}^{z'} = n \) (by the semantics)

- \( t'' = h_i(t_1, \ldots, t_k) \) for \( f_i \in \mathcal{F} \):

\[
|h_i(t_1, \ldots, t_k)|_{w'}^{p''} = n
\]

iff \( w[h_i(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|_{w'}^{p''} \) (by the semantics)

iff \( w[h_i(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by induction)

iff \( w''[h_i(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by Definition 3.31)

iff \( w'[h_i(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by assumption)

iff \( w'_\Sigma[h_i(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|_{w'_\Sigma}^{z'} \) (by Definition 3.31)

iff \( |h_i(t_1, \ldots, t_k)|_{w'_\Sigma}^{z'} = n \) (by the semantics)
A.1 Proof of Theorem 3.39

\[ t'' = g(t_1, \ldots, t_k), \quad \text{where } g \not\in F \cup D \cup \{h_1, \ldots, h_m\}: \]

\[ |g(t_1, \ldots, t_k)|^{|p \cdot z'|} = n \]

if \( w[g(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|^{|p \cdot z'|} \) (by the semantics)

if \( w[g(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|^{|z'|_{w\Sigma}} \) (by induction)

if \( w''[g(n_1, \ldots, n_k), p \cdot z'] = n \), where \( n_j = |t_j|^{|z'|_{w\Sigma}} \) (by Definition 3.31)

if \( w''[g(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|^{|z'|_{w\Sigma}} \) (by assumption)

if \( w''[g(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|^{|z'|_{w\Sigma}} \) (since \( w'' \sim p_{w\Sigma} w' \))

if \( w''[g(n_1, \ldots, n_k), z'] = n \), where \( n_j = |t_j|^{|z'|_{w\Sigma}} \) (by Definition 3.31)

if \( |g(t_1, \ldots, t_k)|^{|z'|_{w\Sigma}} = n \) (by the semantics)

\[ \phi = F_i(t_1, \ldots, t_k), \quad F_i \in F: \]

\( - z' = \langle \rangle : \)

\[ w, p \models F_i(t_1, \ldots, t_k) \]

if \( w[F_i(n_1, \ldots, n_k), p] = 1 \), where \( n_j = |t_i|^{|p \cdot w|} \) (by the semantics)

if \( w[F_i(n_1, \ldots, n_k), p] = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by induction)

if \( w = \gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a^1 \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (since \( w \models \Sigma_{pos} \))

if \( w = \gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (since \( p = |t_i|_{w\Sigma}^{|0 \cdot w|} \) by assumption)

if \( w'' = R[\gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a] \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by Lemma 3.32 (4))

if \( w'' = \gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by Lemma 3.32 (5))

if \( w'' = \langle \gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a \rangle = F_{t_i}(n_1, \ldots, n_k) \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by (A.5))

if \( w'' = F_i(n_1, \ldots, n_k), \langle \rangle = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (since \( w'' \models \langle \rangle \))

if \( w'' \models F_i(n_1, \ldots, n_k), \langle \rangle = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by Definition 3.31)

if \( w''[F_i(n_1, \ldots, n_k), \langle \rangle] = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (by the semantics)

\(- z' = z'' \cdot p': \)

\[ w, p \cdot z'' \cdot p' \models F_i(t_1, \ldots, t_k) \]

if \( w[F_i(n_1, \ldots, n_k), p \cdot z'' \cdot p'] = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|p \cdot z'' \cdot p'|} \) (by the semantics)

if \( w[F_i(n_1, \ldots, n_k), p \cdot z'' \cdot p'] = 1 \), where \( n_j = |t_i|_{w\Sigma}^{|z'' \cdot p'|} \) (by induction)

if \( w, p \cdot z'' = \gamma_{F_i}^{x_1 \cdot \ldots \cdot x_n} a \), where \( n_j = |t_i|_{w\Sigma}^{|0 \cdot w|} \) (since \( w \models \Sigma_{pos} \))
iff $w'_{\Sigma}, z'' \models \gamma F_{n_1 \ldots n_k \ p', \ a}$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma} [F_i (n_1, \ldots, n_k), z'' \cdot p'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}, z'' \cdot p' \models F(t_1, \ldots, t_k)$

(by induction)

(by the semantics)

• $\phi = D(t_1, \ldots, t_k), D \in D$:

iff $w, p \cdot z' \models D(t_1, \ldots, t_k)$

iff $w[D(n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w \cdot z'}$

iff $w[D(n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w, p \cdot z' = \varphi_{Dn_1 \ldots n_k}$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}, z' \models \varphi_{Dn_1 \ldots n_k}$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}[D(n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}, z' \models D(t_1, \ldots, t_k)$

(by the semantics)

(by the semantics)

• $\phi = P_i (t_1, \ldots, t_k)$ for $F_i \in \mathcal{F}$:

iff $w, p \cdot z' \models P_i (t_1, \ldots, t_k)$

iff $w[P_i (n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w \cdot z'}$

iff $w[P_i (n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''[P_i (n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''[P_i (n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}[P_i (n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}, z' \models P_i (t_1, \ldots, t_k)$

(by Definition 3.31)

(by assumption)

(by Definition 3.31)

(by the semantics)

• $\phi = G(t_1, \ldots, t_k), G \notin \mathcal{F} \cup \mathcal{D} \cup \{P_1, \ldots, P_i\}$:

iff $w, p \cdot z' \models G(t_1, \ldots, t_k)$

iff $w[G(n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w \cdot z'}$

iff $w[G(n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''[G(n_1, \ldots, n_k), p \cdot z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''[G(n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''[G(n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w''_{\Sigma}[G(n_1, \ldots, n_k), z'] = 1$, where $n_j = |t_j|_{w'_{\Sigma}}$

iff $w'_{\Sigma}, z' \models G(t_1, \ldots, t_k)$

(by Definition 3.31)

(by assumption)

(by Definition 3.31)

(by the semantics)
A.1 Proof of Theorem 3.39

\[ \phi = (t_1 = t_2): \]

\[ w, p \cdot z' \models (t_1 = t_2) \]

iff \[ |t_1|^{p \cdot z'}_w = |t_2|^{p \cdot z'}_w \] (by the semantics)

iff \[ |t_1|^{z'}_{w \cdot z} = |t_2|^{z'}_{w \cdot z} \] (by induction)

iff \[ w \cdot z', z' \models (t_1 = t_2) \] (by the semantics)

\[ \phi = \phi_1 \wedge \phi_2: \]

\[ w, p \cdot z' \models \phi_1 \wedge \phi_2 \]

iff \[ w, p \cdot z' \models \phi_1 \text{ and } w, p \cdot z' \models \phi_2 \] (by the semantics)

iff \[ w \cdot z', z' \models \phi_1 \text{ and } w \cdot z', z' \models \phi_2 \] (by induction)

iff \[ w \cdot z', z' \models \phi_1 \wedge \phi_2 \] (by the semantics)

\[ \phi = \neg \phi_1: \]

\[ w, p \cdot z' \models \neg \phi_1 \]

iff \[ w, p \cdot z' \not\models \phi_1 \] (by the semantics)

iff \[ w \cdot z', z' \not\models \phi_1 \] (by induction)

iff \[ w \cdot z', z' \models \neg \phi_1 \] (by the semantics)

\[ \phi = \forall x \phi_1: \]

\[ w, p \cdot z' \models \forall x \phi_1 \]

iff \[ w, p \cdot z' \models \phi_{1 \vec{n}} \text{ for all } n \in \mathcal{N}_x \] (by the semantics)

iff \[ w \cdot z', z' \models \phi_{1 \vec{n}} \text{ for all } n \in \mathcal{N}_x \] (by induction)

iff \[ w \cdot z', z' \models \forall x \phi_1 \] (by the semantics)

“\( \leftarrow \)”: Let \( w \) be a world such that \( w \models \Sigma \) and \( w_p = w \cdot z \), where \( p = |t|^{j}_w \). We have to show that \( w' \models \Sigma' \), or equivalently that there exists some \( w'' \) with \( w'' \sim_{P_{\vec{n}}} w' \) such that \( w'' \models \Psi \). Let \( w'' \) be a world that is like \( w' \), except that for every \( P_i \), every \( f_i \), and all names \( \vec{n} \),

\[ w''[P_i(\vec{n}),()] = w[F_i(\vec{n}),()], \]

\[ w''[h_i(\vec{n}),()] = w[f_i(\vec{n}),()]. \]
Obviously $w'' \sim \vec{p}, \vec{h} w'$. In order to show that $w'' \models \Psi$, we first need the property that for every fluent formula $\phi$,

\[(A.12) \quad w \models \phi \iff w'' \models \phi_{\vec{p}, \vec{h}}.\]

We begin by proving the according property for fluent terms $t''$:

\[(A.13) \quad |t''\rangle^0_w = |t''_{\vec{h}}\rangle^0_{w''}.\]

This is done again by an induction over the structure of $t''$:

- $t'' = n \in \mathcal{N}: |t''\rangle^0_w = n = |t''_{\vec{h}}\rangle^0_{w''}$.
- $t'' = f_i(t_1, \ldots, t_k), f_i \in \mathcal{F}$:

\[
|f_i(t_1, \ldots, t_k)\rangle^0_w = w[f_i(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by the semantics)}
\]

\[
= w''[h_i(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by assumption)}
\]

\[
= w''[h_i(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_{\vec{h}} \quad \text{(by induction)}
\]

\[
= |h_i(t^f_{\vec{h}}; \ldots, t^f_{\vec{h}})\rangle^0_{w''} \quad \text{(by the semantics)}
\]

\[
= |f_i(t_1, \ldots, t_k)_{\vec{h}}\rangle^0_{w''} \quad \text{(by definition of } \vec{h})
\]

- $t'' = g(t_1, \ldots, t_k), g \text{ rigid}$:

\[
|g(t_1, \ldots, t_k)\rangle^0_w = w[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by the semantics)}
\]

\[
= w[g(n_1, \ldots, n_k), p], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by the rigidity constraint)}
\]

\[
= w[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by Definition 3.35, and since } w_p = w'_g)\]

\[
= w'[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(by Definition 3.31)}
\]

\[
= w'[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_w \quad \text{(since } w'' \sim \vec{p}, \vec{h} w')
\]

\[
= w'[g(n_1, \ldots, n_k), \langle \rangle], \text{ where } n_j = |t_j\rangle^0_{\vec{h}} \quad \text{(by induction)}
\]

\[
= |g(t^f_{\vec{h}}; \ldots, t^f_{\vec{h}})\rangle^0_{w''} \quad \text{(by the semantics)}
\]

\[
= |g(t_1, \ldots, t_k)_{\vec{h}}\rangle^0_{w''} \quad \text{(by definition of } \vec{h})
\]

Now we can show (A.12) by induction on $\phi$:
A.1 Proof of Theorem 3.39 263

• \( \phi = (t_1 = t_2) \):

\[ w \models (t_1 = t_2) \]

iff \( |t_1|_w = |t_2|_w \) (by the semantics)

iff \( |t_1|_{k_{u''}} = |t_2|_{k_{u''}} \) (by \((A.13))\)

iff \( w'' \models (t_1 = t_2) \) (by the semantics)

iff \( w'' \models (t_1 = t_2) \) (by definition of \( \vec{f} \vec{f} \))

• \( \phi = F_i(t_1, \ldots, t_k) \), \( F_i \in \mathcal{F} \):

\[ w \models F_i(t_1, \ldots, t_k) \]

iff \( w[F_i(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_w \) (by the semantics)

iff \( w''[P_i(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_w \) (by assumption)

iff \( w''[P_i(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_{k_{u''}} \) (by \((A.13))\)

iff \( w'' \models P_i(t_1, \ldots, t_k) \) (by the semantics)

iff \( w'' \models F_i(t_1, \ldots, t_k) \) (by definition of \( \vec{f} \vec{f} \))

• \( \phi = G(t_1, \ldots, t_k) \), \( G \) rigid:

\[ w \models G(t_1, \ldots, t_k) \]

iff \( w[G(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_w \) (by the semantics)

iff \( w[G(n_1, \ldots, n_k), p] = 1 \), where \( n_j = |t_j|_w \) (by the rigidity constraint)

iff \( w'_2[G(n_1, \ldots, n_k), p] = 1 \), where \( n_j = |t_j|_{w_2} \) (by Definition 3.35, and since \( w_p = w'_2 \))

iff \( w''[G(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_w \) (by Definition 3.31)

iff \( w''[G(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_w \) (since \( w'' \sim \vec{P}, \vec{h} w'' \))

iff \( w''[G(n_1, \ldots, n_k), \emptyset] = 1 \), where \( n_j = |t_j|_{k_{u''}} \) (by \((A.13))\)

iff \( w'' \models G(t_1, \ldots, t_k) \) (by the semantics)

iff \( w'' \models G(t_1, \ldots, t_k) \) (by definition of \( \vec{f} \vec{f} \))

• \( \phi = \phi_1 \land \phi_2 \):

\[ w \models \phi_1 \land \phi_2 \]

iff \( w \models \phi_1 \) and \( w \models \phi_2 \) (by the semantics)
iff \( w'' \models \phi_1 F f \vec{P} f \vec{h} \) and \( w \models \phi_2 F f \vec{P} f \vec{h} \) (by induction)

iff \( w'' \models \phi_1 F f \vec{P} f \vec{h} \land \phi_2 F f \vec{P} f \vec{h} \) (by the semantics)

iff \( w'' \models \langle \phi_1 \land \phi_2 \rangle F f \vec{P} f \vec{h} \) (by definition of \( F f \vec{P} f \vec{h} \))

\[ \phi = \neg \phi_1: \]

iff \( w \not\models \phi_1 \) (by the semantics)

iff \( w'' \not\models \phi_1 F f \vec{P} f \vec{h} \) (by induction)

iff \( w'' \models \neg (\phi_1 F f \vec{P} f \vec{h}) \) (by the semantics)

iff \( w'' \models (\neg \phi_1) F f \vec{P} f \vec{h} \) (by definition of \( F f \vec{P} f \vec{h} \))

\[ \phi = \forall x \phi_1: \]

iff \( w \models \forall x \phi_1 \) (by the semantics)

iff \( w'' \models \langle \phi_1 x \rangle F f \vec{P} f \vec{h} \) for all \( n \in \mathcal{N}_x \) (by induction)

iff \( w'' \models \langle \phi_1 x \rangle F f \vec{P} f \vec{h} \) (by (A.7))

iff \( w'' \models \forall x (\phi_1 F f \vec{P} f \vec{h}) \) (by the semantics)

iff \( w'' \models (\forall x \phi_1) F f \vec{P} f \vec{h} \) (by definition of \( F f \vec{P} f \vec{h} \))

We are now ready to prove that \( w \models \Psi: \)

\[ \bullet \quad w'' \models (\Sigma_0) F f \vec{P} f \vec{h}: \]

This follows directly from the fact that \( w \models \Sigma_0 \) and (A.12).

\[ \bullet \quad w'' \models \forall \vec{x}_i (F_i(\vec{x}) \equiv (\gamma F_i) \vec{P} f \vec{h}'); \]

Let \( \vec{n} \) be arbitrary standard name instances for the \( \vec{x}_i \). We have that

\[ w'' \models F_i(\vec{n}) \]

iff \( w''[F_i(\vec{n}), \langle \rangle] = 1 \) (by the semantics)

iff \( w'[F_i(\vec{n}), \langle \rangle] = 1 \) (by assumption)

iff \( w'[\Sigma_i F_i(\vec{n}), \langle \rangle] = 1 \) (by Definition 3.31)

iff \( w_p[F_i(\vec{n}), \langle \rangle] = 1 \) (since \( w_p = w'_\Sigma \) by assumption)
iff $w[F_i(\vec{n}), p] = 1$ (by Definition 3.35)
iff $w, p \models F_i(\vec{n})$ (by the semantics)
iff $w \models \gamma F_i \vec{x}_i^a_p$ (since $w \models \Sigma_{post}$ by assumption)
iff $w \models \gamma F_i \vec{x}_i^a_t$ (since $p = t_{\vec{n}}^t$ by assumption)
iff $w'' \models ((\gamma F_i \vec{x}_i^a_{t_p})_{\vec{n}})$ (by (A.7))

• $w'' \models \forall \vec{x}_j \forall y_i (f_i(\vec{x}_j) = y_i \equiv (\gamma f_i^a_{t_p})_{\vec{n}})$:
Let $\vec{n}$ and $n$ be arbitrary standard name instances for the $\vec{x}_j$ and $y_i$, respectively. We have that

$w'' \models f_i(\vec{n}) = n$
iff $w''[f_i(\vec{n}), \langle \rangle] = n$ (by the semantics)
iff $w'[f_i(\vec{n}), \langle \rangle] = n$ (by Definition 3.31)
iff $w''_\Sigma[f_i(\vec{n}), \langle \rangle] = n$ (by Definition 3.35)
iff $w'[f_i(\vec{n}), \langle \rangle] = n$ (by assumption)
iff $w, p \models f_i(\vec{n}) = n$ (by the semantics)
iff $w \models \gamma f_i \vec{x}_j^a_{n_p}$ (since $w \models \Sigma_{post}$ by assumption)
iff $w \models \gamma f_i \vec{x}_j^a_{n_t}$ (since $p = t_{\vec{n}}^t$ by assumption)
iff $w'' \models (\gamma f_i \vec{x}_j^a_{n_p})_{\vec{n}}$ (by (A.12))
iff $w'' \models ((\gamma f_i^a_{t_p})_{\vec{n}})$ (by (A.7))
A.2 Proof of Lemma 5.19

Lemma 5.19 Let δ be a program (possibly with free variables), w ∈ W, and z ∈ Z. Then for all δ', δ'' appearing in nodes of Gδ (including δ itself) and all variable maps θ', θ'':

1. \langle\langle δ', \cdot \rangle, θ', w, z \rangle \xrightarrow{Gδ} \langle\langle δ'', \cdot \rangle, θ'', w, z \cdot p \rangle \iff \langle z, δ'θ' \rangle \xrightarrow{w} \langle z \cdot p, δ''θ'' \rangle

2. \langle\langle δ', \cdot \rangle, θ', w, z \rangle ∈ F^Gδ \iff \langle z, δ'θ' \rangle ∈ F^w

Proof. We prove the lemma by an induction on the structure of δ. Let Gδ = \langle V, E, v_0 \rangle.

- δ = t:

  1. Since by Definition 5.16, there is only that single edge in G_t, by Definition 5.15 \langle\langle t, \bot \rangle, θ', w, z \rangle \xrightarrow{G_t} \langle\langle nil, \top \rangle, θ', w, z \cdot p \rangle is the only transition step in G_t, where p = |tθ'|_w. Similarly, \langle z, tθ' \rangle \xrightarrow{w} \langle z \cdot p, nil \rangle is the only transition for t according to Definition 5.5. Note that obviously nil = nilθ'.

  2. \langle\langle t, \bot \rangle, θ', w, z \rangle \notin F(Gδ) since w, z \notin \bot according to Definition 5.16, and also \langle z, tθ' \rangle \notin F^w according to Definition 5.5. Furthermore, \langle\langle nil, \top \rangle, θ', w, z \rangle \in F^Gδ since w, z \models \top, and also \langle z, nil \rangle \in F^w.

- δ = α?:

  1. According to Definition 5.16, G_α? does not contain any edges, hence there is no transition possible in the graph, and neither is there for \langle z, α?θ' \rangle according to Definition 5.5.

  2. We have \langle\langle α?, α \rangle, θ', w, z \rangle \in F^Gδ iff w, z \models αθ' (by Definition 5.16) iff \langle z, α?θ' \rangle \in F^w (by Definition 5.5).

- δ = δ₁; δ₂:

  Let G_δ₁ = \langle V_1, E_1, v^1_0 \rangle and G_δ₂ = \langle V_2, E_2, v^2_0 \rangle. For this item, we will need the following consequences of our assumption that all variables are distinct, where for any program δ', let \vec{x} \not∈ δ' mean that none of the \vec{x} appears freely in δ' (similar for formulas):

  (A.14) If \nu \xrightarrow{δ/π\vec{x}_1/ϕ'_1} \nu' \in E_1, then \vec{x} \not∈ δ₂.

  (A.15) If \nu \xrightarrow{δ/π\vec{x}_2/ϕ'_2} \nu' \in E_2, then \vec{x} \not∈ δ₁.

  (A.16) If \vec{x} \not∈ δ₁ and \langle δ'_1, \varphi'_1 \rangle \in V_1, then \vec{x} \not∈ \varphi'_1.

It is easy to check that the variables that are π-quantified in the edges of a program’s characteristic graph are exactly those variables that are π-quantified in the program. By
our assumption, the \( \pi \)-quantified variables in one subprogram are distinct from the \( \pi \)-quantified variables in the other subprogram as well as from any free variables.\footnote{The subprograms may share free variables, though.} Clearly, if \( \vec{x} \not\in \delta' \), then \( \delta''[\vec{x}/\vec{n}] = \delta' \) (similar for formulas).

\((1): \langle \langle \delta', \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta_1}} \langle \delta'', \cdot \rangle, \theta'', w, z \cdot p \rangle \)

iff \( \langle \delta', \cdot \rangle \xrightarrow{\phi_{\pi\vec{x} \bot} / \delta'} \langle \delta'', \cdot \rangle \in E \),
\[ \theta'' = \theta'[\vec{x}/\vec{n}], \ p = |t\theta''|_{w}, \ w, z \models \phi \theta' \wedge \phi \theta'' \] (by Def. 5.15)

iff \( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2 ; \delta_2, \ \langle \delta'_1, \cdot \rangle \xrightarrow{\phi_1 / \pi \vec{x} \bot \delta'_1} \langle \delta'_1, \cdot \rangle \in E_1 \),
\[ \theta'' = \theta'[\vec{x}/\vec{n}], \ \delta_2 \theta'' = \delta_2 \theta', \ p = |t\theta''|_{w}, \ w, z \models \phi_1 \theta' \wedge \phi_2 \theta'' \] or
\( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2 ; \delta_2, \ \langle \delta_1, \cdot \rangle \xrightarrow{\pi \vec{x} \bot / \delta'_2} \langle \delta_1, \cdot \rangle \in E_2, \ \langle \delta', \varphi'_1 \rangle \in V_1, \ \varphi'_1 \not\equiv \bot \),
\[ \theta'' = \theta'[\vec{x}/\vec{n}], \ p = |t\theta''|_{w}, \ w, z \models (\varphi'_1 \wedge \phi_2 \theta''), \ \varphi'_1 \theta'' = \varphi'_1 \theta' \] or
\( \delta' = \delta'_1, \ \delta'' = \delta'_2, \ \langle \delta'_2, \cdot \rangle \xrightarrow{\phi_2 / \pi \vec{x} \bot \delta'_1} \langle \delta''_2, \cdot \rangle \in E_2, \)
\[ \theta'' = \theta'[\vec{x}/\vec{n}], \ p = |t\theta''|_{w}, \ w, z \models \phi_2 \theta' \wedge \phi_2 \theta'' \] (by Def. 5.16 and (A.14) - (A.16))

iff \( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2 ; \delta_2, \ \langle \langle \delta'_1, \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta_1}} \langle \langle \delta'_2, \cdot \rangle, \theta'', w, z \cdot p \rangle \),
\[ \delta_2 \theta'' = \delta_2 \theta' \] or
\( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2, \ \langle \langle \delta_2, \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta_2}} \langle \langle \delta''_2, \cdot \rangle, \theta'', w, z \cdot p \rangle \) (by Def. 5.15)

iff \( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_1 ; \delta_2, \ \langle z, \delta'_1 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_1 \theta'' \rangle, \ \delta_2 \theta'' = \delta_2 \theta' \) or
\( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_1 ; \delta_2, \ \langle z, \delta_2 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_2 \theta'' \rangle, \ \langle z, \delta'_1 \theta' \rangle \in F^w \) or
\( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2, \ \langle z, \delta_2 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_2 \theta'' \rangle \) (by induction)

iff \( \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_1 ; \delta_2, \ \langle z, \delta'_1 ; \delta_2 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_1 ; \delta_2 \theta'' \rangle, \ \delta'_1 ; \delta_2, \ \delta'' = \delta'_2, \ \langle z, \delta'_1 ; \delta_2 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_2 \theta'' \rangle \) or
\[ \delta' = \delta'_1 ; \delta_2, \ \delta'' = \delta'_2, \ \langle z, \delta_2 \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta'_2 \theta'' \rangle \) (by Def. 5.5)

iff \( \langle z, \delta' \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta' \theta'' \rangle \)

\((2): \langle \langle \delta', \cdot \rangle, \theta', w, z \rangle \in F^{G_{\delta}} \)

iff \( w, z \models \varphi(\langle \delta', \cdot \rangle) \theta' \) (by Def. 5.15)

iff \( \delta' = \delta'_1 ; \delta_2, \ \langle \delta'_1, \varphi'_1 \rangle \in V_1, \ \nu'_{\delta_1} = \langle \delta_1, \varphi'_{\delta_1} \rangle, \ w, z \models (\varphi'_1 \wedge \varphi'_2) \theta' \) or
\( \delta' = \delta'_2, \ \langle \delta'_2, \varphi'_2 \rangle \in V_2, \ w, z \models \varphi'_2 \theta' \) (by Def. 5.16)
iff $\delta' = \delta'_1; \delta_2, \langle\langle\delta'_1, \cdot, \theta', w, z\rangle\rangle \in F^{G_{\delta_1}}, \langle\langle\delta_2, \cdot, \theta', w, z\rangle\rangle \in F^{G_{\delta_2}}$ or

$\delta' = \delta'_2, \langle\langle\delta'_2, \cdot, \theta', w, z\rangle\rangle \in F^{G_{\delta_2}}$ (by Def. 5.15)

iff $\delta' = \delta'_1; \delta_2, \langle\langle z, \delta'_1 \theta'\rangle\rangle \in F^w, \langle\langle z, \delta_2 \theta'\rangle\rangle \in F^w$ or

$\delta' = \delta'_2, \langle\langle z, \delta'_2 \theta'\rangle\rangle \in F^w$ (by induction)

iff $\delta' = \delta'_1; \delta_2, \langle\langle z, (\delta'_1; \delta_2) \theta'\rangle\rangle \in F^w$ or

$\delta' = \delta'_2, \langle\langle z, \delta'_2 \theta'\rangle\rangle \in F^w$ (by Def. 5.5)

iff $\langle\langle z, \delta' \theta'\rangle\rangle \in F^w$

• $\delta = \delta_1|\delta_2$:

Let $G_{\delta_1} = (V_1, E_1, v_1^0)$ and $G_{\delta_2} = (V_2, E_2, v_2^0)$.

(1): $\langle\langle \delta', \cdot, \theta', w, z\rangle\rangle \xrightarrow{G_{\delta_1}} \langle\langle \delta'', \cdot, \theta'', w, z \cdot p\rangle\rangle$

iff $\langle\langle \delta', \cdot \rangle\rangle \xrightarrow{\phi/\pi z_i/\phi'_i} \langle\langle \delta''', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi' \land \phi''$ (by Def. 5.15)

iff $\delta' = \delta_1|\delta_2, \theta'' = \delta'_i, \langle\langle \delta_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta_i', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ or

$\delta' = \delta'_i, \theta'' = \delta_i', \langle\langle \delta'_i, \cdot \rangle\rangle \xrightarrow{\phi_i/\pi z_i/\phi'_i} \langle\langle \delta''', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by Def. 5.16)

iff $\delta' = \delta_1|\delta_2, \theta'' = \delta'_i, \langle\langle \delta_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta_i', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by Def. 5.15)

iff $\delta' = \delta_1|\delta_2, \theta'' = \delta_i', \langle\langle \delta_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta_i', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by Def. 5.16)

iff $\langle\langle \delta'_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta''', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by induction)

iff $\delta' = \delta_1|\delta_2, \theta'' = \delta_i', \langle\langle \delta_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta_i', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by Def. 5.5)

iff $\langle\langle \delta'_i, \cdot \rangle\rangle \xrightarrow{\pi z_i/\phi'_i} \langle\langle \delta''', \cdot \rangle\rangle \in E_i, \theta'' = \theta'[\vec{z}/\vec{i}], p = |t\theta''|^z_w, w, z \models \phi_i \theta''$ (by Def. 5.15)

(2): $\langle\langle \delta', \cdot, \theta', w, z\rangle\rangle \in F^{G_{\delta}}$

iff $w, z \models \varphi((\delta', \cdot ))\theta'$ (by Def. 5.15)

iff $\delta' = \delta_1|\delta_2, v_0^1 = \langle\delta_i, \varphi_0^1\rangle, w, z \models \varphi_0^1 \theta'$ or

$\delta' = \delta'_i, \langle\langle \delta'_i, \varphi'_i \rangle\rangle \in E_i, w, z \models \varphi'_i \theta'$ (by Def. 5.16)

iff $\delta' = \delta_1|\delta_2, \langle\langle \delta_i, \cdot \rangle\rangle \in E_i, \theta', w, z \models \varphi_i \theta'$ (by Def. 5.15)
A.2 Proof of Lemma 5.19

Let $\delta$:

- $\delta = \pi y.\delta_1$:

Let $G_{\delta_1} = \langle V_1, E_1, v_0^1 \rangle$.

1: $\langle \langle \delta', \cdot, \theta', w, z \rangle, \theta'' w, z \cdot p \rangle \xrightarrow{G_{\delta_1}} \langle \langle \delta'', \cdot, \theta'' w, z \cdot p \rangle$ (by Def. 5.15)

if $\langle \delta', \cdot \rangle \xrightarrow{\phi/\pi \vec{x}/\vec{n'}} \langle \delta'', \cdot \rangle 
\theta'' = \theta'[\vec{x}/\vec{n}], p = t\theta''_{\vec{x}/\vec{n}}, w, z \models \phi\theta' \land \phi\theta''$

(by induction)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta'_1$, $\langle \delta_1, \cdot \rangle \xrightarrow{\pi \vec{x}/\vec{n'}} \langle \delta'_1, \cdot \rangle \in E_1$

$\theta'' = \theta'[y/n][\vec{x}/\vec{n}] = \theta'[\vec{x}/\vec{n}][y/n], p = t\theta''_{\vec{x}/\vec{n}}, w, z \models \phi_1\theta''$

(by Def. 5.16)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta'_1$, $\langle \delta_1, \cdot \rangle \xrightarrow{\pi \vec{x}/\vec{n'}} \langle \delta'_1, \cdot \rangle \in E_1$

$\theta'' = \theta'[\vec{x}/\vec{n}], p = t\theta''_{\vec{x}/\vec{n}}, w, z \models \phi_1\theta' \land \phi_1\theta''$

(by Def. 5.15)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta'_1$, $\langle \delta_1, \cdot \rangle \xrightarrow{w} \langle z, \delta_1\theta'' \rangle$

(by induction)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta'_1$, $\langle \delta_1, \cdot \rangle \xrightarrow{w} \langle z, \delta_1\theta'' \rangle$

(2): $\langle \langle \delta', \cdot, \theta', w, z \rangle \in F_{G_{\delta_1}}$

if $\langle \langle \delta', \cdot, \theta', w, z \rangle \in F_{G_{\delta_1}}$

(by Def. 5.15)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta_1$, $\langle \delta_1, \cdot \rangle \in V_1, w, z \models \varphi(\delta', \cdot)$

(by Def. 5.16)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta_1$, $\langle \delta_1, \cdot \rangle \in V_1, w, z \models \varphi_0\theta'$

(by the semantics)

if $\delta' = \pi y.\delta_1$, $\delta'' = \delta_1$, $\langle \delta_1, \cdot \rangle \in \delta_1, w, z \models \varphi_0\theta'$

(by Def. 5.15)
If $\delta' = \pi y, \delta_1$, $\langle z, \delta_1, \theta' \rangle \in F^w$ or

$$\delta' = \delta_1', \langle z, \delta_1', \theta' \rangle \in F^w$$

(by induction)

If $\delta' = \pi y, \delta_1$, $\langle z, (\pi y, \delta_1)' \theta' \rangle \in F^w$ or

$$\delta' = \delta_1', \langle z, \delta_1', \theta' \rangle \in F^w$$

(by Def. 5.5)

If $\langle z, \delta', \theta' \rangle \in F^w$

- $\delta = \delta_1 \parallel \delta_2$:
  Let $G_{\delta_1} = \langle V_1, E_1, v_1^0 \rangle$ and $G_{\delta_2} = \langle V_2, E_2, v_2^0 \rangle$. Here, we make use of the following consequences of our distinctiveness assumption for $\pi$-quantified variables:

(A.17) If $v \xrightarrow{\phi / \pi \vec{x}/ \pi \vec{y}/ \phi'} v' \in E_1$ and $\langle \delta_2', \cdot \rangle \in V_2$, then $\vec{x} \not\in \delta_2'$.

(A.18) If $v \xrightarrow{\phi / \pi \vec{x}/ \pi \vec{y}/ \phi'} v' \in E_2$ and $\langle \delta_1', \cdot \rangle \in V_1$, then $\vec{x} \not\in \delta_1'$.

Again, if $\vec{x} \not\in \delta_1'$, then $\delta_{\vec{x}} = \delta_1'$.

(1) $\langle \langle \delta', \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta}} \langle \langle \delta'', \cdot \rangle, \theta'', w, z \cdot p \rangle$

If $\langle \delta', \cdot \rangle \xrightarrow{\phi / \pi \vec{x}/ \pi \vec{y}/ \phi'} \langle \delta'', \cdot \rangle \in E_1$,

$$\theta'' = \theta'[\vec{x}/ \vec{y}], p = |\theta''|_{w}, w, z \models \phi \theta' \land \phi \theta''$$

(by Def. 5.15)

If $\delta' = \delta_1' \parallel \delta_2$, $\theta'' = \delta_1' \parallel \delta_2$, $\langle \delta_1', \cdot \rangle \xrightarrow{\phi / \pi \vec{x}/ \pi \vec{y}/ \phi'} \langle \delta_1'', \cdot \rangle \in E_1$,

$$\theta'' = \theta'[\vec{x}/ \vec{y}], \delta_1'' \theta'' = \delta_1 \theta', p = |\theta''|_{w}, w, z \models \phi_1 \theta' \land \phi_1' \theta'' \land \phi_2 \theta'' \lor$$

(by Def. 5.16 and (A.17), (A.18))

If $\delta' = \delta_1' \parallel \delta_2$, $\theta'' = \delta_1' \parallel \delta_2$, $\langle \langle \delta_1', \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta_1}} \langle \langle \delta_1'', \cdot \rangle, \theta'', w, z \cdot p \rangle$,

$$\delta_1'' \theta'' = \delta_1' \theta' \lor$$

If $\delta' = \delta_1' \parallel \delta_2$, $\theta'' = \delta_1' \parallel \delta_2$, $\langle \langle \delta_1', \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{\delta_2}} \langle \langle \delta_1'', \cdot \rangle, \theta'', w, z \cdot p \rangle$,

$$\delta_1'' \theta'' = \delta_1' \theta'$$

(by Def. 5.15)

If $\delta' = \delta_1' \parallel \delta_2$, $\theta'' = \delta_1' \parallel \delta_2$, $\langle z, (\delta_1', \theta') \rangle \xrightarrow{w} \langle z \cdot p, (\delta_1', \theta') \rangle$, $\delta_1'' \theta'' = \delta_1' \theta' \lor$

(by induction)

If $\delta' = \delta_1' \parallel \delta_2$, $\theta'' = \delta_1' \parallel \delta_2$, $\langle z, (\delta_1', \theta') \rangle \xrightarrow{w} \langle z \cdot p, (\delta_1', \theta') \rangle$, $\delta_1'' \theta'' = \delta_1' \theta'$

(by Def. 5.5)

If $\langle z, \delta', \theta' \rangle \xrightarrow{w} \langle z \cdot p, \delta' \theta' \rangle$
A.2 Proof of Lemma 5.19

(2): \( \langle \delta', \cdot \rangle, \theta', w, z \rangle \in \mathcal{F}^{G_b} \)

iff \( w, z \models \varphi(\langle \delta', \cdot \rangle)\theta' \) (by Def. 5.15)

iff \( \delta' = \delta'_1 \| \delta'_2, (\delta'_1, \varphi'_1) \in V_1, (\delta'_2, \varphi'_2) \in V_2, w, z \models (\varphi'_1 \land \varphi'_2)\theta' \) (by Def. 5.16)

iff \( \delta' = \delta'_1 \| \delta'_2, (\delta'_1, \cdot, \theta', w, z) \in \mathcal{F}^{G_{b1}}, (\delta'_2, \cdot, \theta', w, z) \in \mathcal{F}^{G_{b2}} \) (by Def. 5.15)

iff \( \delta' = \delta'_1 \| \delta'_2, (z, \delta'_1 \theta') \in \mathcal{F}^w, (z, \delta'_2 \theta') \in \mathcal{F}^w \) (by induction)

iff \( (z, \delta' \theta') \in \mathcal{F}^w \)

• \( \delta = (\delta_1)^* \):

Let \( G_{b1} = \langle V_1, E_1, v_1 \rangle \). Again we need two properties for variable substitution, following from the assumption that the \( \pi \)-quantified variables are distinct from the free variables of a program:

(A.19) If \( v \xrightarrow{\phi/\pi x/\phi'} v' \in E_1 \), then \( \bar{x} \notin \delta_1 \).

(A.20) If \( \bar{x} \notin \delta_1 \), then \( \bar{x} \notin (\delta_1)^* \).

(1): \( \langle \langle \delta', \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{b1}} \langle \langle \delta'', \cdot \rangle, \theta'', w, z \cdot p \rangle \)

iff \( \langle \delta', \cdot \rangle \xrightarrow{\phi/\pi x/\phi'} \langle \delta'', \cdot \rangle \in E_1 \)

\[ \theta'' = \theta'[\bar{x}/\bar{n}], \quad p = \mid t\theta'' \mid_w, \quad w, z \models \phi\theta' \land \phi\theta'' \]

(by Def. 5.15)

iff \( \delta' = (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle \delta'_1, \cdot \rangle \xrightarrow{\pi x; \delta'_1} \langle \delta'_1, \cdot \rangle \in E_1 \),

\[ \theta'' = \theta'[\bar{x}/\bar{n}], \quad (\delta_1)^* \theta'' = (\delta_1)^* \theta', \quad p = \mid t\theta'' \mid_w, \quad w, z \models \phi\theta' \land \phi\theta'' \]

or

iff \( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle \delta'_1, \varphi'_1 \rangle \in E_1, \langle \delta_1, \cdot \rangle \xrightarrow{\pi x; \delta'_1} \langle \delta'_1, \cdot \rangle \in E_1 \),

\[ \theta'' = \theta'[\bar{x}/\bar{n}], \quad (\delta_1)^* \theta'' = (\delta_1)^* \theta', \quad p = \mid t\theta'' \mid_w, \quad w, z \models \phi\theta' \land \phi\theta'' \]

(by Def. 5.16 and (A.19), (A.19))

iff \( \delta' = (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle \delta_1, \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{b1}} \langle \langle \delta'_1, \cdot \rangle, \theta'', w, z \cdot p \rangle \),

\( (\delta_1)^* \theta'' = (\delta_1)^* \theta', \) or

iff \( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle \delta'_1, \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{b1}} \langle \langle \delta'_1, \cdot \rangle, \theta'', w, z \cdot p \rangle \),

\( (\delta_1)^* \theta'' = (\delta_1)^* \theta', \) or

iff \( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle \delta_1, \cdot \rangle, \theta', w, z \rangle \xrightarrow{G_{b1}} \langle \langle \delta'_1, \cdot \rangle, \theta'', w, z \cdot p \rangle \),

\( \langle \delta'_1, \cdot \rangle, \theta', w, z \rangle \in \mathcal{F}^{G_{b1}}, \quad (\delta_1)^* \theta'' = (\delta_1)^* \theta', \)

(by Def. 5.15)
(2): \( \langle \langle \delta', \cdot \rangle, \theta', w, z \rangle \in F^{G_{\delta_1}} \)

- if \( w, z \models \varphi((\delta', \cdot))\theta' \) (by Def. 5.15)
- if \( \delta' = (\delta_1)^* \), \( w, z \models \top \) or
  \( \delta' = \delta'_1; (\delta_1)^*, \langle \delta'_1, \varphi'_1 \rangle \in V_1, w, z \models \varphi'_1 \theta' \) (by Def. 5.16)
- if \( \delta' = (\delta_1)^* \), \( w, z \models \top \) or
  \( \delta' = \delta'_1; (\delta_1)^*, \langle \langle \delta'_1, \cdot \rangle, \theta', w, z \rangle \in F^{G_{\delta_1}} \) (by Def. 5.15)
- if \( \delta' = (\delta_1)^* \), \( w, z \models \top \) or
  \( \delta' = \delta'_1; (\delta_1)^*, \langle \delta_1 \theta'; (\delta_1)^* \theta' \rangle \in F^w \) (by induction)
- if \( \delta' = (\delta_1)^* \), \( z, (\delta_1)^* \) \( \in F^w \) or
  \( \delta' = \delta'_1; (\delta_1)^*, \langle z, (\delta_1)^* \rangle \theta' \) \( \in F^w \) (by Def. 5.5)
- if \( \langle z, \delta' \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, \delta^* \theta'' \rangle \)

\( \delta' = (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, \delta_1 \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, \delta'_1 \theta'' \rangle \)

\( (\delta_1)^* \theta'' = (\delta_1)^* \theta' \) or

\( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, \delta_1 \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, \delta'_1 \theta'' \rangle \)

\( (\delta_1)^* \theta'' = (\delta_1)^* \theta' \) or

\( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, \delta_1 \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, \delta'_1 \theta'' \rangle \)

\( \langle z, \delta'_1; (\delta_1)^* \rangle \in F^w, (\delta_1)^* \theta'' = (\delta_1)^* \theta' \) (by induction)

\( \delta' = (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, (\delta_1)^* \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, (\delta'_1; (\delta_1)^*) \theta'' \rangle \) or

\( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, (\delta'_1; (\delta_1)^*) \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, (\delta'_1; (\delta_1)^*) \theta'' \rangle \) or

\( \delta' = \delta'_1; (\delta_1)^*, \delta'' = \delta'_1; (\delta_1)^*, \langle z, (\delta'_1; (\delta_1)^*) \theta' \rangle \stackrel{w}{\rightarrow} \langle z \cdot p, (\delta'_1; (\delta_1)^*) \theta'' \rangle \)

\( \langle z, (\delta'_1; (\delta_1)^*) \theta' \rangle \in F^w \) (by Def. 5.5)
A.3 Proof of Lemma 5.59

Lemma 5.59 Let $w \in W$ and $\Sigma = \Sigma_0 \cup \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$ a basic action theory over $\langle D, F \rangle$ such that $w \models \Sigma_{\text{def}} \cup \Sigma_{\text{post}}$. Furthermore let $\varphi \in \mathcal{E}\mathcal{S}_{\text{LTL}}$ and $G = \langle V, E, v_0 \rangle$ a trace graph. The following are equivalent:

1. There is some $w_0 \approx w$ with $w_0, z \models \varphi \downarrow \theta$ and a run
   \[ (v_0, \theta, w_0, z) \xrightarrow{G, \varphi} (v_1, \theta_1, w_1, z \cdot p_1) \xrightarrow{G, \varphi} \cdots \]
   such that for all $k \geq 0$ exists $l \geq k$ with $w_1, z \cdot p_1 \cdots p_l \models \text{AccAll}[\varphi]$.

2. $w, z, p_1, p_2, \cdots \models \varphi \theta$ and there is a run
   \[ (v_0, \theta, w, z) \xrightarrow{G} (v_1, \theta_1, w, z \cdot p_1) \xrightarrow{G} (v_2, \theta_2, w, z \cdot p_1 \cdot p_2) \xrightarrow{G} \cdots \]

Proof. We prove the two directions separately. In the following, let $\pi = p_1 \cdot p_2 \cdots$. We will abbreviate the sequence $p_1 \cdots p_k$ as $z_k$ and use $\pi_k$ to denote $p_{k+1} \cdot p_{k+2} \cdots$ for any $k \geq 0$.

"1 \Rightarrow 2": First we establish the existence of the run. For this purpose, we show by induction on $k$:

(A.21) If $\langle v, \theta, w, z \rangle \xrightarrow{G, \varphi} \langle v_1, \theta_1, w_1, z \cdot z_1 \rangle \xrightarrow{G, \varphi} \cdots \xrightarrow{G, \varphi} \langle v_k, \theta_k, w_k, z \cdot z_k \rangle$, then $w_k \approx w$ and $\langle v_0, \theta, w, z \rangle \xrightarrow{G} \langle v_1, \theta_1, w, z \cdot z_1 \rangle \xrightarrow{G} \cdots \xrightarrow{G} \langle v_k, \theta_k, w, z \cdot z_k \rangle$.

- $k = 0$:
  $w_0 \approx w$ by assumption.

- $k \mapsto k + 1$:
  Now assume that $\langle v, \theta, w, z \rangle \xrightarrow{G, \varphi} \cdots \xrightarrow{G, \varphi} \langle v_k, \theta_k, w_k, z \cdot z_k \rangle$ and that $\langle v_k, \theta_k, w_k, z \cdot z_k \rangle \xrightarrow{G, \varphi} \langle v_{k+1}, \theta_{k+1}, w_{k+1}, z \cdot z_{k+1} \rangle$. By induction, $\langle v_0, \theta, w, z \rangle \xrightarrow{G} \cdots \xrightarrow{G} \langle v_k, \theta_k, w, z \cdot z_k \rangle$ and $w_k \approx w$. By Definition 5.56, we get $\langle v_k, \theta_k, w_k, z \cdot z_k \rangle \xrightarrow{G} \langle v_{k+1}, \theta_{k+1}, w_{k+1}, z \cdot z_{k+1} \rangle$ and $w_{k+1} \approx w$. Therefore $w_{k+1} \approx w$ by Proposition 5.53, and thus also $w_{k+1} \approx w$ by Proposition 5.52. $\langle v_k, \theta_k, w, z \cdot z_k \rangle \xrightarrow{G} \langle v_{k+1}, \theta_{k+1}, w, z \cdot z_{k+1} \rangle$ then follows from Definition 5.15 and the fact that $w_k$ and $w$ must agree on all edge conditions as they do not contain any of the auxiliary predicates.

Next, we need the following property of runs according to item 1:

(A.22) For all $k \geq 0$ and all $\theta'$, there exists $l \geq k$ such that $w_1, z \cdot z_l \models \text{Accept}_i \downarrow \theta'$.

To see why, suppose not. Then there is some $k \geq 0$, some $i$ and some $\theta'$ such that for all $l' \geq k$, $w_{l'}, z \cdot z_{l'} \not\models \text{Accept}_i \downarrow \theta'$. Since by assumption $w_{l'+1}, z \cdot z_{l'} \models \Sigma_{\text{post}}(p_{l'+1})$, we get with
Moreover, by assumption we have that there must be one \( l_1 \geq k \) with \( w_{l_1}, z \cdot z_{l_1} \models \text{AccAll}[\varphi] \), and that there is some \( l_2 > l_1 \) such that also \( w_{l_2}, z \cdot z_{l_2} \models \text{AccAll}[\varphi] \). Let both \( l_1 \) and \( l_2 \) be the smallest indexes with these properties. Since \( \text{AccAll}[\varphi] \) entails \( A_i(\vec{x}_i)\theta' \), and because by assumption \( w_{l_3}, z \cdot z_{l_3} \not\models \text{AccAll}[\varphi] \) for all \( l_3 \) with \( k \leq l_3 < l_2 \) and \( l_3 \neq l_1 \), we get that \( w_{l_2-1}, z \cdot z_{l_2-1} \models A_i(\vec{x}_i)\theta' \) from (A.23) and \( w_{l_2} \approx_{\vec{x}_i, \vec{u}, \vec{A}} w_{l_2-1} \). By repeatedly applying the same argument, we obtain that this holds for all indexes \( l_4 \) with \( l_2 > l_4 > l_1 \). Due to \( w_{l_1}, z \cdot z_{l_1} \models \text{AccAll}[\varphi] \), (A.23) and \( w_{l_1+1} \approx_{\vec{x}_i, \vec{u}, \vec{A}} w_{l_1} \) implies that \( w_{l_1+1}, z \cdot z_{l_1+1} \not\models A_i(\vec{x}_i)\theta' \), contradiction.

We can now turn to the satisfaction of \( \varphi \). For that matter, we prove the following more general property by an induction on its structure: For all subformulas \( \phi \) of \( \varphi \), all variable maps \( \theta' \) and all \( k \geq 0 \),

\[
(A.24) \quad w_k, z \cdot z_k \models \phi_\downarrow \theta' \text{ iff } w, z \cdot z_k, \pi_k \models \phi \theta'
\]

- The base cases \( \phi = (t_1 = t_2) \) and \( \phi = F(\vec{v}) \) follow immediately since \( \phi_\downarrow = \phi \), \( w_k \approx_{\vec{x}_i, \vec{u}, \vec{A}} w \) and \( \phi \) does not contain any of the \( \vec{x}_i, \vec{u}, \vec{A} \).

- Furthermore, the cases \( \phi = \neg \theta' \), \( \phi = \phi_1 \wedge \phi_2 \) and \( \exists x. \phi' \) follow directly by induction.

- \( \phi = X \phi_j: \)

  \[
  w_k, z \cdot z_k \models X \phi_j_\downarrow \theta'
  \]

  iff \( w_k, z \cdot z_k \models X_j(\vec{x}_i)\theta' \) \hspace{1cm} (by Definition 5.48)

  iff \( w_{k+1}, z \cdot z_k \models X_j(\vec{x}_i)\theta' \) \hspace{1cm} (since \( w_{k+1} \approx_{\vec{x}_i, \vec{u}, \vec{A}} w_k \))

  iff \( w_{k+1}, z \cdot z_k \models [p_{k+1}]_i \phi_j_\downarrow \theta' \) \hspace{1cm} (by (5.91) and since \( w_{k+1}, z \cdot z_k \models \text{Trans}[\varphi]_{\pi_{k+1}} \))

  iff \( w_{k+1}, z \cdot z_k \cdot p_{k+1} \models \phi_j_\downarrow \theta' \) \hspace{1cm} (by the semantics)

  iff \( w, z \cdot z_k \cdot p_{k+1}, \pi_{k+1} \models \phi_j \theta' \) \hspace{1cm} (by induction)

  iff \( w, z \cdot z_k, \pi_k \models X \phi_j \theta' \) \hspace{1cm} (by the semantics)

- \( \phi = (\phi_i U \psi_i): \) Then \( \phi_\downarrow \theta' = U_i(\vec{x}_i)\theta' \). We prove the two directions separately.

  \( \Rightarrow: \) Let \( w_k, z \cdot z_k \models U_i(\vec{x}_i)\theta' \) and \( l_0 \geq k \) be the smallest index such that \( w_{l_0}, z \cdot z_{l_0} \models \text{Accept}_{i,l_0} \theta' \), which must exist according to (A.22). Then for all \( l' \) with \( k \leq l' < l_0 \), \( w_{l'}, z \cdot z_{l'} \not\models \text{Accept}_{i,l'} \theta' \). By (A.22), this means that for all such \( l' \), \( w_{l'}, z \cdot z_{l'} \models U(\vec{x}_i)\theta' \land \neg \psi_i \downarrow \theta' \).

  As \( w_{l'}, z \cdot z_{l'} \models \text{LocCons}[\varphi]_\downarrow \), we then furthermore get \( w_{l'}, z \cdot z_{l'} \models \phi_{i,l'} \theta' \) by (5.89). This
together with the facts that \( w_{l_0}, z \cdot z_{l_0-1} \models \Trans_0[\varphi] \) and \( w_{l_0} \overset{z \cdot z_l}{\approx} X, \emptyset, \bar{A} \) \( w_{l_0-1} \) implies that \( w_{l_0}, z \cdot z_{l_0} \models U(\bar{x}_1) \theta' \) using (5.90), and that hence \( w_{l_0}, z \cdot z_{l_0} \models \psi_1 \downarrow \theta' \) by (5.92). By induction then \( w, z \cdot z_l, \pi_l \models \phi \theta' \) for all \( l \) with \( k \leq l < l_0 \), and \( w, z \cdot z_{l_0}, \pi_{l_0} \models \psi_1 \theta' \). By the semantics, this means that \( w, z \cdot z_k, \pi_k \models (\phi_1 \ U \ \psi_1) \theta' \).

"\( \Leftarrow \)" Let \( w, z \cdot z_k, \pi_k \models (\phi_1 \ U \ \psi_1) \theta' \). By the semantics, there then is some \( l \geq k \) such that \( w, z \cdot z_l, \pi_l \models \psi_1 \theta' \) and for all \( l' \) with \( k \leq l' < l \), \( w, z \cdot z_{l'}, \pi_{l'} \models \phi_1 \theta' \). Let \( l \) further be the smallest such index. By induction, we obtain \( w_1, z \cdot z_l \models \psi_1 \Downarrow \theta' \) and that for all \( l' \) with \( k \leq l' < l \), \( w, z \cdot z_{l'}, \pi_{l'} \models \phi_1 \theta' \). Since \( w_{l+1} \overset{z \cdot z_{l+1}}{\approx} X, \emptyset, \bar{A} \) \( w_1 \) and \( w_{l+1}, z \cdot z_l \models \LocCons[\varphi] \), also \( w_1, z \cdot z_l \models \LocCons[\varphi] \). Hence by (5.88) we get \( w_1, z \cdot z_l \models U_1(\bar{x}_1) \theta' \). Repeated applying (5.90), and using the facts that \( w_{l+1} \overset{z \cdot z_{l+1}}{\approx} X, \emptyset, \bar{A} \) \( w_1 \) and that \( w_{l+1}, z \cdot z_{l'} \models \Trans_0[\varphi] \), yields \( w_{l'}, z \cdot z_{l'} \models U_1(\bar{x}_1) \theta' \) for all \( l' \) with \( k \leq l' < l \), including the case \( l' = k \).

By (A.24) we now get in particular that if \( w_0, z \models \varphi \theta \), then \( w, z, \pi \models \varphi \theta \).

"1 \( \Rightarrow \) 2": First we construct a sequence of \( w_0, w_1, w_2, \ldots \), given the original \( w \). Let \( w_0 \) be world that is like \( w \), except that for all \( k \geq 0 \), all \( i \), all \( j \), and all \( \bar{n} \)

\[
\begin{align*}
  w_0[X_j(\bar{n}), z \cdot z_k] &= 1 \text{ iff } w, z \cdot z_k, \pi_k \models X_f \bar{n}^j \\
  w_0[U_i(\bar{n}), z \cdot z_k] &= 1 \text{ iff } w, z \cdot z_k, \pi_k \models U_\psi \bar{n}^i \\
  w_0[A_i(\bar{n}), z] &= 0 \\
  w_0[A_i(\bar{n}), z \cdot z_{k+1}] &= 1 \text{ iff } w_0, z \cdot z_k \models \gamma_{A\bar{n}} \bar{n}^a p_{k+1}
\end{align*}
\]

Obviously \( w_0 \overset{z \cdot z_k}{\approx} X, \emptyset, \bar{A} \) \( w \). Moreover, we simply set \( w_k = w_0 \) for all \( k > 0 \). Then trivially, for all \( k \geq 0 \), \( w_{k+1} \overset{z \cdot z_{k+1}}{\approx} X, \emptyset, \bar{A} \) \( w_k \), and \( w_{k+1}, z \cdot z_k \models \Sigma^A_{\text{pos}}(p_{k+1}) \).

We furthermore note that for all subformulas \( \phi \) of \( \varphi \), all variable maps \( \theta' \) and all \( k \geq 0 \),

\[
A.25 \quad w_k, z \cdot z_k \models \phi_1 \theta' \text{ iff } w, z \cdot z_k, \pi_k \models \phi \theta'
\]

which follows by a simple induction over the strucute of \( \phi \) and our construction of the \( w_k \).

Hence in particular, \( w_0, z \models \varphi \downarrow \theta \).

Next, we need to establish that for all \( k \geq 0 \), \( \langle v_k, \theta_k, w_k, z \cdot z_k \rangle \overset{\varphi}{\rightarrow} \langle v_{k+1}, \theta_{k+1}, w_{k+1}, z \cdot z_{k+1} \rangle \).

According to Definition 5.56, this requires the following:

- \( \langle v_k, \theta_k, w_k, z \cdot z_k \rangle \overset{\varphi}{\rightarrow} \langle v_{k+1}, \theta_{k+1}, w_{k+1}, z \cdot z_{k+1} \rangle \)

By assumption, \( \langle v_k, \theta_k, w, z \cdot z_k \rangle \overset{\varphi}{\rightarrow} \langle v_{k+1}, \theta_{k+1}, w, z \cdot z_{k+1} \rangle \), and since \( w_k = w_0 \overset{z \cdot z_k}{\approx} X, \emptyset, \bar{A} \) \( w \), they in particular agree on all edge conditions as they do not contain any of the \( X, \emptyset, \bar{A} \).

Therefore \( \langle v_k, \theta_k, w_k, z \cdot z_k \rangle \overset{\varphi}{\rightarrow} \langle v_{k+1}, \theta_{k+1}, w_{k+1}, z \cdot z_{k+1} \rangle \).
\[ w_{k+1}, z \cdot z_k \models \text{LocCons}[\varphi] \downarrow: \]

- (5.88):
  
  Let \( w_0, z \cdot z_k \models \psi_i \downarrow \theta' \). By (A.25), \( w, z \cdot z_k, \pi_k \models \psi_i \theta' \), therefore by the semantics \( w, z \cdot z_k, \pi_k \models (\phi_i U \psi_i) \theta' \), hence again by (A.25) \( w_0, z \cdot z_k \models U(\bar{x}_i) \theta' \).

- (5.89):
  
  Let \( w_0, z \cdot z_k \models U_i(\bar{x}_i) \theta' \land -\psi_i \downarrow \theta' \). By (A.25), \( w, z \cdot z_k, \pi_k \models (\phi_i U \psi_i) \theta' \land -\psi_i \theta' \), therefore by the semantics \( w, z \cdot z_k, \pi_k \models \phi_i \theta' \), hence again by (A.25) \( w_0, z \cdot z_k \models \phi_i \downarrow \theta' \).

- (5.90):
  
  \[ w_0, z \cdot z_k \models X_j(\bar{x}_j) \theta' \]
  
  iff \( w, z \cdot z_k \models X \phi_j \theta' \) (by construction)
  
  iff \( w, z \cdot z_{k+1} \models \phi_j \theta' \) (by the semantics)
  
  iff \( w_0, z \cdot z_{k+1} \models \phi_j \downarrow \theta' \) (by (A.25))
  
  iff \( w_0, z \cdot z_k \models [p_{k+1}] \phi_j \downarrow \theta' \) (by the semantics)

- (5.91):
  
  \( w_0, z \cdot z_k \models U_i(\bar{x}_i) \theta' \)
  
  iff \( w, z \cdot z_k, \pi_k \models (\phi_i U \psi_i) \theta' \) (by (A.25))
  
  iff \( w, z \cdot z_k, \pi_k \models \psi_i \theta' \) or \( w, z \cdot z_k, \pi_k \models \phi_i \theta' \)
  
  and \( w, z \cdot z_{k+1}, \pi_{k+1} \models (\phi_i U \psi_i) \theta' \) (by the semantics)
  
  iff \( w_0, z \cdot z_k \models \psi_i \downarrow \theta' \) or \( w_0, z \cdot z_k \models \phi_i \downarrow \theta' \) and \( w_0, z \cdot z_{k+1} \models U(\bar{x}_i) \theta' \) (by (A.25))
  
  iff \( w_0, z \cdot z_k \models \psi_i \downarrow \theta' \land \phi_i \downarrow \theta' \land [p_{k+1}] U (\bar{x}_i) \theta' \) (by the semantics)

Finally, the property remains to be shown that for all \( k \geq 0 \) there is some \( l \geq k \) such that \( w_0, z \cdot z_l \models \text{AccAll}[\varphi] \). Suppose not, then there is some smallest \( k \geq 0 \) such that for all \( l \geq k \), \( w_0, z \cdot z_l \not\models \text{AccAll}[\varphi] \). Using that \( w_0, z \cdot z_l \models \Sigma_{\text{post}}(p_{l+1}) \), this implies that for all such \( l \geq k \) and all \( i \),

\[
(A.26) \quad w_0, z \cdot z_l \models [p_{l+1}] A_i(\bar{x}_i) \equiv (\text{Accept}_{i \downarrow} \lor A_i(\bar{x}_i))
\]

according to (5.101). Furthermore, suppose that \( k > 0 \), then \( w_0, z \cdot z_{k-1} \models \text{AccAll}[\varphi] \), and by (A.26) also \( w_0, z \cdot z_k \models \text{AccAll}[\varphi] \), contradicting our assumption. Therefore it must be that \( k = 0 \). Note that by (A.26), the extensions of the \( A_i \) are monotonically non-decreasing: Once
A.3 Proof of Lemma 5.59

$A_i(\vec{n})$ gets true in $w_0, z_{l+1}$ due to $w_0, z \cdot z_l \models \text{Accept}_{i, \vec{x}_n}$, it will be that $A_i(\vec{n})$ will hold in all $w_0, z_{l'}$ with $l' \geq l$. There is thus at least some $i$ and some $\vec{n}$ such that $\text{Accept}_{i, \vec{x}_n}$ never holds in any $w_0, z_{l'}$. By (A.25), this means that for all $l' \geq 0$, $w, z \cdot z_{l'} \models (\phi_i \ U \psi_i)^\vec{x}_{l'} \land \neg \psi_i^\vec{x}_{l'}$, which is impossible in our semantics. Contradiction. \qed
Appendix B

Examples

B.1 Complete Example 5.35

\[ L_0 = \{ \langle v_0, \varphi \rangle, \langle v_1, \varphi \rangle, \langle v_2, \varphi \rangle \} \]

First Iteration

- **PRE**[\(v_0, L_0\)]
  \[
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]\varphi] \lor \\
  R[\exists x. \neg \text{Empty}(\text{queue}) \land \text{Poss}(\text{selectRequest}(x)) \land [\text{selectRequest}(x)]\varphi] \lor \\
  R[\text{Empty}(\text{queue}) \land \text{Poss}(\text{wait}) \land [\text{wait}]\varphi] \equiv \\
  \exists a \exists x'(a = \text{requestCoffee}(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land T) \lor \\
  \exists x(\neg \text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(\text{queue}, x) \land \bot) \lor \\
  \text{Empty}(\text{queue}) \land T \land T \equiv \\
  \text{LastFree}(\text{queue}) \lor \bot \lor \text{Empty}(\text{queue})
  \]

\[
\equiv \exists x_1(\text{queue} = \langle x_1, e \rangle) \lor \text{queue} = \langle e, e \rangle \\
\equiv \exists x_1(\text{queue} = \langle x_1, e \rangle) \\
\equiv \text{LastFree}(\text{queue})
\]

- **PRE**[\(v_1, L_0\)]
  \[
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a]\varphi] \lor \ R[\text{Poss}(\text{pickupCoffee}) \land [\text{pickupCoffee}]\varphi] \equiv \\
  \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land T) \lor \neg \text{HoldingCoffee} \land T \equiv \\
  \text{LastFree}(\text{queue}) \lor \neg \text{HoldingCoffee}
  \]
\[ \text{Pre}[v_2, L_0] \]
\[ \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree}(\text{queue}))] \lor R[\text{Poss}(\text{bringCoffee}(x)) \land [\text{bringCoffee}(x)]\varphi] \]
\[ \equiv \exists a \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top) \lor \text{HoldingCoffee} \land \top \]
\[ \equiv \text{LastFree}(\text{queue}) \lor \text{HoldingCoffee} \]

\[ L_1 = L_0 \text{ AND Pre}[G_0, L_0] \]
\[ = \{ (v_0, \varphi \land \text{LastFree}(\text{queue})), (v_1, \varphi \land (\text{LastFree}(\text{queue}) \lor \neg HC)), (v_2, \varphi \land (\text{LastFree}(\text{queue}) \lor HC)) \} \]

\text{Second Iteration}

\[ \text{Pre}[v_0, L_1] \]
\[ \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree}(\text{queue}))] \lor R[\exists x. \neg \text{Empty}(\text{queue}) \land \text{Poss}(\text{sR}(x)) \land [sR(x)](\varphi \land (\text{LastFree}(\text{queue}) \lor \neg HC)))] \lor R[\text{Empty}(\text{queue}) \land \text{Poss}(\text{wait}) \land [\text{wait}](\varphi \land \text{LastFree}(\text{queue}))] \]
\[ \equiv \exists a \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top \land \exists x_1 \text{Enqueue}(\text{queue}, x', (x_1, e)) \lor \exists x(\neg \text{Empty}(\text{queue}) \land x \neq e \land \text{IsFirst}(\text{queue}, x) \land \bot \land (\cdots) \land \text{Empty}(\text{queue}) \land \top \land \text{LastFree}(\text{queue}) \land \top) \]
\[ \equiv \text{LastFree}(\text{queue}) \land \exists x' \exists x_1 \text{queue} = (e, e) \land x' = x_1 \land \bot \land \text{Empty}(\text{queue}) \]
\[ \equiv \text{Empty}(\text{queue}) \lor \bot \lor \text{Empty}(\text{queue}) \]
\[ \equiv \text{Empty}(\text{queue}) \]

\[ \text{Pre}[v_1, L_1] \]
\[ \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land (\text{LastFree}(\text{queue}) \lor \neg HC))] \lor R[\text{Poss}(\text{pickupCoffee}) \land [\text{pickupCoffee}](\varphi \land (\text{LastFree}(\text{queue}) \lor HC))] \]
\[ \equiv \exists a \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree}(\text{queue}) \land \top \land (\exists x_1 \text{Enqueue}(\text{queue}, x', (x_1, e)) \lor \neg HC) \lor \neg HC \land \top \land (\text{LastFree}(\text{queue}) \lor \top) \]
\[ \equiv \text{LastFree}(\text{queue}) \land (\text{Empty}(\text{queue}) \lor \neg HC) \lor \neg HC \]
\[ \equiv \text{Empty}(\text{queue}) \lor \neg \text{HoldingCoffee} \]
• \text{Pre}[v_2, L_1]
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land (\text{LastFree(queue)} \lor HC))] \lor
  R[\text{Poss}(\text{bringCoffee}(x)) \land [\text{bringCoffee}(x)](\varphi \land \text{LastFree(queue)})]
  \equiv \exists a \exists x'(a = rC(x') \land x' \neq e \land \text{LastFree(queue)} \land \top \land (\exists x_1 \text{Enqueue(queue, x', (x_1, e)}) \lor HC) \lor
  HC \land \top \land \text{LastFree(queue)}
  \equiv \text{LastFree(queue)} \land (\text{Empty(queue)} \lor HC) \lor HC \land \text{LastFree(queue)}
  \equiv \text{LastFree(queue)} \land (\text{Empty(queue)} \lor \text{HoldingCoffee})

L_2 = L_1 \text{ AND Pre}[g_3, L_1]
= \{\langle v_0, \varphi \land \text{LastFree(queue)} \land \text{Empty(queue)}\rangle,\ 
  \langle v_1, \varphi \land (\text{LastFree(queue)} \lor \neg HC) \land (\text{Empty(queue)} \lor \neg HC)\rangle,\ 
  \langle v_2, \varphi \land (\text{LastFree(queue)} \land HC) \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor HC)\rangle \}
\equiv \{\langle v_0, \varphi \land \text{Empty(queue)}\rangle,\ 
  \langle v_1, \varphi \land (\text{Empty(queue)} \lor \neg HC)\rangle,\ 
  \langle v_2, \varphi \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor HC)\rangle \}

\text{Third Iteration}

• \text{Pre}[v_0, L_2]
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty(queue)})] \lor
  R[\exists r. \neg \text{Empty(queue)} \land \text{Poss}(sR(x)) \land [sR(x)](\varphi \land (\text{Empty(queue)} \lor \neg HC))] \lor
  R[\text{Empty(queue)} \land \text{Poss(wait)} \land [\text{wait}](\varphi \land \text{Empty(queue)})]
  \equiv \bot \lor \bot \lor \text{Empty(queue)}
  \equiv \text{Empty(queue)}

• \text{Pre}[v_1, L_2]
  \equiv R[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land (\text{Empty(queue)} \lor \neg HC))] \lor
  R[\text{Poss}(pC) \land [pC](\varphi \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor HC))]\]
  \equiv \text{LastFree(queue)} \land \top \land (\bot \lor \neg HC) \lor
  \neg HC \land \top \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor \top)
  \equiv \text{LastFree(queue)} \land \neg HC \lor \neg HC \land \text{LastFree(queue)}
  \equiv \text{LastFree(queue)} \land \neg \text{HoldingCoffee}
\textbf{Pre}[^{v_2}, L_2] \\
\equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor \text{HC}))] \lor \\
\mathcal{R}[\text{Poss(bringCoffee}(x)) \land [\text{bringCoffee}(x)](\varphi \land \text{LastFree(queue)})] \\
\equiv \text{LastFree(queue)} \land \top \land \text{Empty(queue)} \land (\bot \lor \text{HC}) \lor \\
\text{HC} \land \top \land \text{LastFree(queue)} \\
\equiv \text{Empty(queue)} \land \text{HC} \lor \text{HC} \land \text{LastFree(queue)} \\
\equiv \text{LastFree(queue)} \land \text{HoldingCoffee}

L_3 = L_2 \text{ AND } \text{Pre}[G_3, L_2] \\
= \{(v_0, \varphi \land \text{Empty(queue)} \land \text{Empty(queue)}), \\
\langle v_1, \varphi \land (\text{Empty(queue)} \lor \neg \text{HC}) \land \text{LastFree(queue)} \land \neg \text{HC} \rangle, \\
\langle v_2, \varphi \land \text{LastFree(queue)} \land (\text{Empty(queue)} \lor \text{HC}) \land \text{LastFree(queue)} \land \text{HC} \rangle \} \\
\equiv \{(v_0, \varphi \land \text{Empty(queue)}), \\
\langle v_1, \varphi \land \text{LastFree(queue)} \land \neg \text{HC} \rangle, \\
\langle v_2, \varphi \land \text{LastFree(queue)} \land \text{HC} \rangle \} \}

Fourth Iteration

\textbf{Pre}[v_0, L_3] \\
\equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty(queue)})] \lor \\
\mathcal{R}[\exists r. \neg \text{Empty(queue)} \land \text{Poss}(sR(x)) \land [sR(x)](\varphi \land \text{LastFree(queue)} \land \neg \text{HC})] \lor \\
\mathcal{R}[\text{Empty(queue)} \land \text{Poss}(\text{wait}) \land [\text{wait}](\varphi \land \text{Empty(queue)})] \\
\equiv \bot \lor \bot \lor \bot \land \text{Empty(queue)} \\
\equiv \text{Empty(queue)} \\

\textbf{Pre}[v_1, L_3] \\
\equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree(queue)} \land \neg \text{HC})] \lor \\
\mathcal{R}[\text{Poss(pC)} \land [pC](\varphi \land \text{LastFree(queue)} \land \text{HC})] \\
\equiv \text{LastFree(queue)} \land \top \land \text{Empty(queue)} \land \neg \text{HC} \lor \\
\neg \text{HC} \land \top \land \text{LastFree(queue)} \land \top \\
\equiv \text{Empty(queue)} \land \neg \text{HC} \lor \neg \text{HC} \land \text{LastFree(queue)} \\
\equiv \text{LastFree(queue)} \land \neg \text{HoldingCoffee}
• \textbf{PRE}[v_2, L_3]
  \begin{align*}
  & \equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree(queue)} \land HC)] \lor \\
  & \mathcal{R}[\text{Poss(bringCoffee}(x)) \land [\text{bringCoffee}(x)](\varphi \land \text{Empty(queue)})]
  \end{align*}
  \begin{align*}
  & \equiv \text{LastFree(queue)} \land \top \land \text{Empty(queue)} \land HC \lor \\
  & HC \land \top \land \text{Empty(queue)}
  \end{align*}
  \begin{align*}
  & \equiv \text{Empty(queue)} \land HC \lor HC \land \text{Empty(queue)}
  \end{align*}
  \begin{align*}
  & \equiv \text{Empty(queue)} \land \text{HoldingCoffee}
  \end{align*}

\[L_4 = L_3 \text{ AND } \text{PRE}[G_0, L_3]\]
\begin{align*}
  & = \{ (v_0, \varphi \land \text{Empty(queue)} \land \text{Empty(queue)}), \\
  & (v_1, \varphi \land \text{LastFree(queue)} \land \neg HC \land \text{LastFree(queue)} \land \neg HC), \\
  & (v_2, \varphi \land \text{LastFree(queue)} \land HC \land \text{Empty(queue)} \land HC) \}
  \end{align*}
\begin{align*}
  & \equiv \{ (v_0, \varphi \land \text{Empty(queue)}), \\
  & (v_1, \varphi \land \text{LastFree(queue)} \land \neg HC), \\
  & (v_2, \varphi \land \text{Empty(queue)} \land HC) \}
  \end{align*}

\textbf{Fifth Iteration}

• \textbf{PRE}[v_0, L_4]
  \begin{align*}
  & \equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty(queue)}))] \lor \\
  & \mathcal{R}[\exists r. \neg \text{Empty(queue)} \land \text{Poss(sR}(x)) \land [sR}(x)](\varphi \land \text{LastFree(queue)} \land \neg HC))] \lor \\
  & \mathcal{R}[\text{Empty(queue)} \land \text{Poss(wait)} \land [\text{wait]}(\varphi \land \text{Empty(queue)}))]
  \end{align*}
  \begin{align*}
  & \equiv \top \lor \top \lor \top \lor \text{Empty(queue)}
  \end{align*}
  \begin{align*}
  & \equiv \text{Empty(queue)}
  \end{align*}

• \textbf{PRE}[v_1, L_4]
  \begin{align*}
  & \equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{LastFree(queue)} \land \neg HC))] \lor \\
  & \mathcal{R}[\text{Poss}(pC) \land [pC](\varphi \land \text{Empty(queue)} \land HC))]
  \end{align*}
  \begin{align*}
  & \equiv \text{LastFree(queue)} \land \top \land \text{Empty(queue)} \land \neg HC \lor \\
  & \neg HC \land \top \land \text{Empty(queue)} \land \top
  \end{align*}
  \begin{align*}
  & \equiv \text{Empty(queue)} \land \neg HC \lor \neg HC \land \text{Empty(queue)}
  \end{align*}
  \begin{align*}
  & \equiv \text{Empty(queue)} \land \neg \text{HoldingCoffee}
  \end{align*}
\[
\text{PRE}[v_2, L_4] \\
\equiv \mathcal{R}[\exists a.\text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty}(\text{queue}) \land \text{HC})] \lor \\
\mathcal{R}[\text{Poss}(\text{bringCoffee}(x)) \land [\text{bringCoffee}(x)](\varphi \land \text{Empty}(\text{queue}))] \\
\equiv \text{LastFree}(\text{queue}) \land \top \land \bot \land \text{HC} \lor \\
\text{HC} \land \top \land \text{Empty}(\text{queue}) \\
\equiv \bot \lor \text{HC} \land \text{Empty}(\text{queue}) \\
\equiv \text{Empty}(\text{queue}) \land \text{HoldingCoffee}
\]

\[
L_5 = L_4 \text{ AND } \text{PRE}[G_8, L_4] \\
= \{ (v_0, \varphi \land \text{Empty}(\text{queue}) \land \text{Empty}(\text{queue})) , \\
(v_1, \varphi \land \text{LastFree}(\text{queue}) \land \neg \text{HC} \land \text{Empty}(\text{queue}) \land \neg \text{HC}) , \\
v_2, \varphi \land \text{Empty}(\text{queue}) \land \text{HC} \land \text{Empty}(\text{queue}) \land \text{HC} \} \\
\equiv \{ (v_0, \varphi \land \text{Empty}(\text{queue})) , \\
(v_1, \varphi \land \text{Empty}(\text{queue}) \land \neg \text{HC} , \\
v_2, \varphi \land \text{Empty}(\text{queue}) \land \text{HC} \} \\
\equiv \bot \lor \neg \text{HC} \land \text{Empty}(\text{queue}) \\
\equiv \text{Empty}(\text{queue}) \land \neg \text{HoldingCoffee}
\]

Sixth Iteration

\[
\text{PRE}[v_0, L_5] \\
\equiv \mathcal{R}[\exists a.\text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty}(\text{queue}))] \lor \\
\mathcal{R}[\exists x.\neg \text{Empty}(\text{queue}) \land \text{Poss}(\text{sR}(x)) \land [\text{sR}(x)](\varphi \land \text{Empty}(\text{queue}) \land \neg \text{HC})] \lor \\
\mathcal{R}[\text{Empty}(\text{queue}) \land \text{Poss}(\text{wait}) \land [\text{wait}](\varphi \land \text{Empty}(\text{queue}))] \\
\equiv \bot \lor \bot \lor \bot \land \text{Empty}(\text{queue}) \\
\equiv \text{Empty}(\text{queue})
\]

\[
\text{PRE}[v_1, L_5] \\
\equiv \mathcal{R}[\exists a.\text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty}(\text{queue}) \land \neg \text{HC})] \lor \\
\mathcal{R}[\text{Poss}(pC) \land [pC](\varphi \land \text{Empty}(\text{queue}) \land \text{HC})] \\
\equiv \text{LastFree}(\text{queue}) \land \top \land \bot \land \neg \text{HC} \lor \\
\neg \text{HC} \land \top \land \text{Empty}(\text{queue}) \land \top \\
\equiv \bot \lor \neg \text{HC} \land \text{Empty}(\text{queue}) \\
\equiv \text{Empty}(\text{queue}) \land \neg \text{HoldingCoffee}
\]
• $\text{PRE}[v_2, L_5]$
  \[\equiv \mathcal{R}[\exists a. \text{Exo}(a) \land \text{Poss}(a) \land [a](\varphi \land \text{Empty}(\text{queue}) \land HC)] \lor \mathcal{R}[\text{Poss}(\text{bringCoffee}(x)) \land [\text{bringCoffee}(x)](\varphi \land \text{Empty}(\text{queue}))])\]
  \[\equiv \text{LastFree}(\text{queue}) \land \top \land \bot \land HC \lor HC \land \top \land \text{Empty}(\text{queue}) \land \bot\]
  \[\equiv \bot \lor HC \land \text{Empty}(\text{queue}) \land \text{HoldingCoffee}\]

$L_6 = L_5 \text{ AND } \text{PRE}[G_5, L_5]$
  \[= \{\langle v_0, \varphi \land \text{Empty}(\text{queue}) \land \text{Empty}(\text{queue})\rangle, \langle v_1, \varphi \land \text{Empty}(\text{queue}) \land \neg HC \land \text{Empty}(\text{queue}) \land \neg HC\rangle, \langle v_2, \varphi \land \text{Empty}(\text{queue}) \land HC \land \text{Empty}(\text{queue}) \land HC\rangle \} \equiv \{\langle v_0, \varphi \land \text{Empty}(\text{queue})\rangle, \langle v_1, \varphi \land \text{Empty}(\text{queue}) \land \neg HC\rangle, \langle v_2, \varphi \land \text{Empty}(\text{queue}) \land HC\rangle \}$
Bibliography


Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In Mark S. Boddy, Maria


[DGKK98] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. TAL: Temporal action logics language specification and tutorial. Electronic


[ENLC06] Patrick Eyerich, Bernhard Nebel, Gerhard Lakemeyer, and Jens Claßen. Golog and PDDL: What is the relative expressiveness? In Xiaoping Chen, Wei Liu,


[KS92] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann,
editor, *Proceedings of the Tenth European Conference on Artificial Intelligence

[KS96] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, proposi-
tional logic and stochastic search. In Dan Weld and Bill Clancey, editors, *Pro-
cedings of the Thirteenth National Conference on Artificial Intelligence (AAAI

[Lak99] Gerhard Lakemeyer. On sensing and offline-interpreting in GOLOG. In *Logical
Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages

[Lak10] Gerhard Lakemeyer. The situation calculus: A case for modal logic. *Journal of

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of

[Lev84] Hector J. Levesque. Foundations of a functional approach to knowledge repre-
and 95.

[Lev90] Hector J. Levesque. All I know: a study in autoepistemic logic. *Artificial

[Lev96] Hector J. Levesque. What is planning in the presence of sensing? In Dan Weld
and Bill Clancey, editors, *Proceedings of the Thirteenth National Conference on
on page 94.

dro Saffiotti, editors, *Proceedings of the Nineteenth International Joint Confer-
ence on Artificial Intelligence (IJCAI 2005)*, pages 509–515. Professional Book


[Rei98] Raymond Reiter. Sequential, temporal GOLOG. In Anthony G. Cohn, Lenhard K. Schubert, and Stuart C. Shapiro, editors, *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and


Cited on page 151.

Cited on page 46.


Index

3APL, 15, 21

A, 15
achieve, 112, 145
action, 27
    atomic, 88
    exogenous, 89, 148, 182
Action Description Language, see ADL
action formalism, see action logic
action language, see action logic
action logic, 11
action occurrence, 183
add list, 15, 128
ADL, 16, 122, 154
ADL atom, 125
ADL domain, 124
ADL effect, 126
ADL ground action, 128
ADL ground atom, 125
ADL object domain, 127
ADL operator, 125
ADL plan, 128
ADL precondition formula, 125
ADL problem, 124
ADL state, 128
ADL truth, 128
ADL update, 128
advice taker, 11
agent, 1

AI, 1
\(\mathcal{A}\), 219, 245
algebraic numbers, 46
Alternating-Time Temporal Logic, see ATL
Answer Set Programming, see ASP
\(\mathcal{AOL}\), 29
arithmetical correctness constraint, 34
Artificial Intelligence, see AI
ASP, 15
ATL, 245
ATL, 21
B, 15
Barcan formula, 80
basic action theory, 57
    wrt an ADL problem, 129
BAT, see basic action theory
BDI, 15
beliefs-desire-intention, see BDI
bounded, 33
Büchi automaton, 219, 220, 224, 245
\(\mathcal{C}\), 15
\(\mathcal{C}+\), 15
Caesar (robot), 1
CASLve, 21
causal completeness assumption, 54, 55
cCGOLOG, 14
characteristic graph, 186
CHECKEG, 203
Index

CheckEU, 209
CheckEX, 199
CheckLTL, 224
CheckPost, 236
closed-world assumption, 114, 120
dynamic, 146
local, 146
Cognitive Robotics, 2
compactness, 36
compilation scheme, 142
computable numbers, 46
concurrency, 89
concurrent iteration, 89
conditional, 88
configuration, 91, 170, 185
ConGolog, 14, 89, 167
connective, 32
consistency condition, 54, 55
countability, 37, 45
counterexample, 246
CTL, 20, 165, 193
CTL*, 20, 165, 219
decidability, 219, 245
Default Logic, 51
delete list, 15, 128
denotation, 34
derived predicates, 122
description logic, 219, 245
diagnosis, 21
domain closure, 114, 143
DTGolog, 14
Dynamic Logic, 15, 19
effect axiom, 52, 55
    negative, 53
    positive, 53
elementary set, 222
emulation, 161
entailment, 35, 80
Epistemic Dynamic Logic, 15
epistemic feasibility, 94, 146
epistemic operator, 77, 247
epistemic state, 78
equality, 37
ES, 4, 29
ESj, 165
ESj\textsubscript{CTL}, 193
ESj\textsubscript{CTL*}, 220
ESj\textsubscript{LTL}, 221
eventual compliance, 223
Event Calculus, 15
executability, 108
    of an ADL plan, 129
exogenous action, see action, exogenous
explanation closure, 54
explanation closure axiom, 55
expressiveness, 4, 142
expressivity, see expressiveness
Fast Downward, 16
Features and Fluents, 15
FF, 16, 145
finality, 91, 170, 185, 227
first-order logic, see FOL
fixpoint, 165, 217
fluent, 27
    definitional, 56
    functional, 27
    normal, 56
    relational, 27
Fluent Calculus, 14, 156
FLUX, 14, 156
FOADD, 242
FOL, 3, 24, 36, 216
forgetting, 234
formula, 31, 78
  basic, 78
  bounded, 33, 169
  fluent, 56
  in regressable form, 62
objective, 31
objective SO, 47
primitive, 31
regressable, 61
situation, 168
static, 33, 169
subjective, 78
trace, 169
  verifiable, 196, 221
foundational axioms, 28
frame axiom, 53
frame problem, 11, 53, 131
function symbol, 30

General Game Playing, 15
General Problem Solver, see GPS
GOAL, 21
Golog, 3, 13, 85
GPS, 15
GraphPlan, 16
  graph configuration, 185
  graph finality, 185, 227
  graph trace, 186
  graph transition, 185, 227
GTGolog, 14
  guarded execution, 94, 148
Hierarchical Task Network, see HTN
Hoare logic, 19, 21, 245
HTN, 18
identity, 26
incompleteness theorem, 24
IndiGolog, 14, 90, 144
induction, 39
infimum, 217
initial database, 57
  wrt an ADL problem, 130
initial knowledge base, see initial database
initial state, 125
initial theory, see initial database
integers, 46
interaction operations, 95
International Planning Competition, see IPC
interrupt, 89
IPC, 5, 16, 113
iteration, 88
join, 217

Knaster-Tarski Fixpoint Theorem, 218
knowledge, 76
  de dicto, 26, 77
  de re, 26, 77
knowledge-based agent, 94
knowledge-based system, 2
Knowledge-Based Systems Group Aachen, 1
knowledge base, 2
  proper\textsuperscript{+}, 157, 219, 246
Knowledge Representation and Reasoning, see KR
Knowledge Representation Hypothesis, 2
KR, 2, 23
INDEX

\( \mathcal{L} \), 36
label, 196
labelling, 196
label equivalence, 197
label operations, 197
lattice
  complete, 217
  well-founded, 218
learning, 107
linear temporal logic, see LTL
local consistency, 222
Logical Omniscience, 4
loop, 88
  infinite, 168
Löwenheim-Skolem theorem, 37
LR-Progression, 71
LTL, 18, 20, 219, 220
macro semantics, 88
Markov Decision Process, see MDP
MDP, 17
meet, 217
model checking, 7, 20, 164
  CTL, 193
  CTL\(^*\), 220
  symbolic, 20, 164
multi-agent system, 107
\( \mathcal{N} \), 30
\( \mathcal{N}_A \), 30
natural numbers, 46
negative effect, 126
\( \mathcal{N}_N \), 30
\( \mathcal{N}_O \), 30
nondeterministic choice, 88
nondeterministic choice of argument, 88
nonmonotonic reasoning, 51
NuSMV, 20
OBDD, 20, 242
  first-order, 242
objective truth, 34
occlusion, 155
offline, 14, 90
online, 14, 90, 94, 144, 146
only-knowing, 78
ordered binary decision diagram, see OBDD
\( \mathcal{P}_A \), 31
Partially Ordered MDP, see POMDP
partial correctness, 245
PATH, 199, 208
\( \text{PATH}_{\text{LTL}} \), 229
PDDL, 5, 16, 116
  temporal, 121, 154
peer reviewing, 161
PKS, 248
planning, 4, 15, 111
  conditional, 17
  conformant, 17
  domain-dependent, 18
  domain-independent, 18
  generalized, 17
\( \mathcal{P}_N \), 31
\( \mathcal{P}_O \), 31
POMDP, 17
positive effect, 126
possible world, 13, 78
precondition, 50
precondition axiom, 57
  wrt an ADL problem, 131
precondition extension, 177
<table>
<thead>
<tr>
<th>Predicate Abstraction</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate Quantifier Elimination</td>
<td>234</td>
</tr>
<tr>
<td>Predicate Symbol</td>
<td>30</td>
</tr>
<tr>
<td>Preferences</td>
<td>17, 122</td>
</tr>
<tr>
<td>Preimage</td>
<td>198, 225</td>
</tr>
<tr>
<td>Prioritized Concurrency</td>
<td>89</td>
</tr>
<tr>
<td>Procedure</td>
<td>248</td>
</tr>
<tr>
<td>Program</td>
<td>91, 167</td>
</tr>
<tr>
<td>Knowledge-based Program</td>
<td>92, 99</td>
</tr>
<tr>
<td>Program Trace</td>
<td>171</td>
</tr>
<tr>
<td>Program Finality</td>
<td>91, 170</td>
</tr>
<tr>
<td>Program Transition</td>
<td>91, 170</td>
</tr>
<tr>
<td>Progression</td>
<td>13, 68, 71, 101, 108, 114, 136</td>
</tr>
<tr>
<td>Of an Epistemic State</td>
<td>79</td>
</tr>
<tr>
<td>Of a World</td>
<td>68</td>
</tr>
<tr>
<td>Projection Problem</td>
<td>12, 60</td>
</tr>
<tr>
<td>Property Persistence</td>
<td>245</td>
</tr>
<tr>
<td>Provability</td>
<td>24</td>
</tr>
<tr>
<td>Qualification Axiom</td>
<td>51</td>
</tr>
<tr>
<td>Qualification Problem</td>
<td>50, 131</td>
</tr>
<tr>
<td>Quantification</td>
<td></td>
</tr>
<tr>
<td>Second-Order</td>
<td>47, 226, 234</td>
</tr>
<tr>
<td>Substitutional</td>
<td>35, 45</td>
</tr>
<tr>
<td>Typed</td>
<td>38</td>
</tr>
<tr>
<td>Rational Numbers</td>
<td>46</td>
</tr>
<tr>
<td>Readylog</td>
<td>14</td>
</tr>
<tr>
<td>Real Numbers</td>
<td>43</td>
</tr>
<tr>
<td>Regression</td>
<td>12, 60, 81, 96</td>
</tr>
<tr>
<td>Regression Operator</td>
<td>62</td>
</tr>
<tr>
<td>Regression Theorem</td>
<td>65</td>
</tr>
<tr>
<td>Relative Completeness</td>
<td>245</td>
</tr>
<tr>
<td>Representation Theorem</td>
<td>84</td>
</tr>
<tr>
<td>Requirement Flags</td>
<td>122</td>
</tr>
<tr>
<td>RES</td>
<td>83</td>
</tr>
<tr>
<td>Rhino (Robot)</td>
<td>1</td>
</tr>
<tr>
<td>Rigid</td>
<td>27</td>
</tr>
<tr>
<td>Rigidity Constraint</td>
<td>34</td>
</tr>
<tr>
<td>Robocup</td>
<td>1</td>
</tr>
<tr>
<td>Run</td>
<td></td>
</tr>
<tr>
<td>Finite</td>
<td>185, 227</td>
</tr>
<tr>
<td>Infinite</td>
<td>186, 227</td>
</tr>
<tr>
<td>Satisfiability</td>
<td>35, 80</td>
</tr>
<tr>
<td>SCAN</td>
<td>234</td>
</tr>
<tr>
<td>Search Operator</td>
<td>90, 146, 157</td>
</tr>
<tr>
<td>Second-Order Compatibility</td>
<td>47</td>
</tr>
<tr>
<td>Second-Order Logic, see SOL</td>
<td></td>
</tr>
<tr>
<td>Semi-Decidability</td>
<td>24</td>
</tr>
<tr>
<td>Sense-Plan-Act, see Sense-Reason-Act</td>
<td></td>
</tr>
<tr>
<td>Sense-Reason-Act</td>
<td>5, 94</td>
</tr>
<tr>
<td>Sensing</td>
<td>78, 151</td>
</tr>
<tr>
<td>Sensing Compatibility</td>
<td>79</td>
</tr>
<tr>
<td>Sensing Fluent Axiom</td>
<td>79</td>
</tr>
<tr>
<td>Sentence</td>
<td>31</td>
</tr>
<tr>
<td>Sequence</td>
<td>88</td>
</tr>
<tr>
<td>sGolog</td>
<td>14</td>
</tr>
<tr>
<td>Shakey (Robot)</td>
<td>1, 12</td>
</tr>
<tr>
<td>SHOP2</td>
<td>18</td>
</tr>
<tr>
<td>SimpleAPL</td>
<td>21</td>
</tr>
<tr>
<td>Simple Golog</td>
<td>87</td>
</tr>
<tr>
<td>Simulation</td>
<td>161</td>
</tr>
<tr>
<td>Single-Step Consistency</td>
<td>223</td>
</tr>
<tr>
<td>Situation</td>
<td>27</td>
</tr>
<tr>
<td>Situation Calculus</td>
<td>3, 11</td>
</tr>
<tr>
<td>S</td>
<td>157</td>
</tr>
<tr>
<td>SCA</td>
<td>157</td>
</tr>
<tr>
<td>SOL</td>
<td>24</td>
</tr>
<tr>
<td>Sort</td>
<td>27, 31</td>
</tr>
<tr>
<td>SPIN</td>
<td>20</td>
</tr>
<tr>
<td>SRI</td>
<td>12</td>
</tr>
</tbody>
</table>
standard name, 26, 30, 108
co-referring, 34
Stanford Research Institute, see SRI
state constraint, 21
state update axiom, 14
static, 33
STRIPS, 4, 12, 15, 114
strongest post-condition, 71
successor state axiom, 12, 54, 56
instantiation, 227
strongly context free, 114
wrt an ADL problem, 131
supremum, 217
symbol grounding, 107
synchronization, 180
syntactical preference, 32
TAL, 15, 18, 155
TALplanner, 18, 155
Temporal Action Logic, see TAL
term, 31, 108
fluent, 56
ground, 31
primitive, 31
test, 88
testing, 161
theorem proving, 25, 246
TLPlan, 18
Toronto Cognitive Robotics Group, 13
trace, 171, 185, 186
admitted by a graph, 186
of an ADL plan, 128
of an infinite run, 186
of a finite run, 185
of a program, 171
trace graph, 184
tractability, 4
trajectory constraints, 122
transfinite iteration, 218
transition, 91, 170, 185, 227
transition semantics, 90, 91, 170, 180
transitive closure, 48
truth, 79
objective, 34
of ADL formulas, 128
of situation formulas, 171
of trace formulas, 172
type, 124
compound, 124
primitive, 124
UCPOP, 16
uncertainty, 107
uncountability, 45
Unifying Action Calculus, 156
unique names for actions, 35
unique names for actions constraint, 34
universal generalization, 39, 235
validity, 35, 80
of an ADL plan, 129
variable, 29
bound, 31
free, 31
variable map, 185
verification, 6, 19, 161
verification transform operator
$\mathcal{E}\mathcal{S}\mathcal{G}_{\text{CTL}}$, 196
$\mathcal{E}\mathcal{S}\mathcal{G}_{\text{CTL}^*}$, 221
$\mathcal{W}$, 34
weak S5, 80
wh-question, 26
witness, 246
world, 33
\mathcal{Z}, 31