Towards More Realistic Logic-Based Robot Controllers in the GOLOG Framework

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen zur
Erlangung des akademischen Grades eines Doktors der
Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Henrik Grosskreutz

aus Berlin

Berichter:

Universitätsprofessor Gerhard Lakemeyer, Ph.D.
Universitätsprofessor Dr. Michael Thielscher

Tag der mündlichen Prüfung: 1. Februar 2002

To my parents

Acknowledgements

First and foremost, I would like to thank my advisor Gerhard Lakemeyer. I particularly enjoyed our discussions, from which I took great benefit. He taught me to scrutinize preliminary conceptualizations in order to clarify and structure my ideas, and had a profound impact on my scientific education. I am also indebted to him for reading and commenting on earlier versions of this thesis. This work wouldn't have been possible without his help and advice.

I would also like to thank Michael Beetz who had a strong influence on my view of high-level robot control. Thanks also to Gero Iwan, who was always interested in the problems I struggled with and who gave me useful suggestions, Frank Dylla, Günter Gans and Alexander Ferrein. I am also grateful to Michael Thielscher who kindly took on the task of second reviewer for this thesis. Finally, I would like to thank Simone for bravely listening to my complaints during the tough times and for enjoying the good times with me, my family and my friends.

Abstract

High-level robot control languages should not only be expressive but should also support reasoning about actions, in particular, the projection of robot plans. Projection is useful for the robot when choosing among different courses of action as well as for the designer of robot controllers, since projections allow for qualitative simulations. The high-level programming language GOLOG was specifically proposed for this purpose. The semantics of GOLOG, which offers constructs such as sequences, iterations and recursive procedures, is based on the situation calculus, a logical language for reasoning about action and change. In particular, every primitive GOLOG action is an action of the underlying situation calculus theory, which allows reasoning about the effects of primitive actions or complex GOLOG programs.

While GOLOG comes equipped with a powerful projection mechanism, however, it lacks the expressiveness provided by non-logic-based robot programming languages like RPL, RAP or Colbert. In particular, it does not provide facilities for dealing with continuous change, event-driven behavior, and communication with lower-level routines for navigation or localization, to which we refer to as *low-level processes*. Another limitation of GOLOG is that it assumes that actions have deterministic effects and cannot represent probabilistic uncertainty. In realistic domains, however, uncertainty seems to be ubiquitous: a robot has often only probabilistic beliefs about the state of the world, and low-level processes have probabilistic outcomes.

In this thesis, we show how the GOLOG framework can be extended to deal with issues like continuous change, event-driven actions and low-level processes in a natural way, thus shortening the gap in expressiveness between non-logic-based and logic-based robot control languages. In particular, we integrate continuous time and change directly in the ontology of GOLOG. To facilitate the actual execution of high-level plans on a real robot, we employ a layered robot control architecture where a high-level controller communicates via message with the low-level processes provided by the basic-task execution system. Our framework allows not only the projection and the actual (on-line) execution of the same plans, but also supports the specification of plans with interleave projection and on-line execution. Furthermore, we provide means to represent and deal with probabilistic beliefs and noisy sensors and effectors. Finally, the extended GOLOG formalism is implemented in PROLOG and evaluated in several experiments, including delivery tasks where the mobile robot CARL operates in the Computer Science Department V at Aachen University of Technology.

Zusammenfassung

Sprachen zur Steuerung mobiler Roboter sollten nicht nur eine geeignete Ausdrucksstärke besitzen, sondern zudem die Möglichkeit bieten, Pläne zu *projizieren*, das heißt automatisch über deren Effekte auf den Zustand der Welt zu folgern. Diese Fähigkeit ist sowohl bei der Auswahl eines geeigneten Plans durch den Roboter als auch während der Entwicklung hilfreich, da sie qualitative Aussagen über die Auswirkungen eines Plans liefert. Aus dieser Motivation heraus wurde die Sprache GOLOG entwickelt, die Programmkonstrukte wie Sequenzen, Iterationen und rekursive Prozeduren bietet und deren Semantik auf dem Situationenkalkül beruht, einem logischen Formalismus zum Schließen über Aktionen und deren Effekte. Insbesondere ist jede primitive GOLOG-Instruktion eine Aktion der zugrundeliegenden Situationenkalkül-Theorie, was das Schließen über die Effekte einzelner Aktionen sowie komplexer Programme ermöglicht.

Dieser herausragenden Eigenschaft steht jedoch die geringe Ausdrucksstärke von GOLOG verglichen mit nicht-logikbasierten Roboterprogrammiersprachen wie RPL, RAP oder Colbert gegenüber. Insbesondere bietet GOLOG keine Kontrollstrukturen zum Umgang mit kontinuierlicher Veränderung, ereignisgesteuerten Aktionen oder zur Kommunikation mit spezialisierten Basisprozessen, die Grundfunktionalitäten wie Navigation oder Lokalisierung realisieren. Zudem verwendet GOLOG ein deterministisches Modell der Effekte von primitiven Aktionen und ermöglicht es nicht, probabilistische Unsicherheit über den Zustand der Welt zu repräsentieren. Diese ist in realistischen Robotikanwendungen jedoch allgegenwärtig.

Ziel dieser Arbeit ist es, die Ausdrucksstärke von GOLOG an die nicht-logikbasierter Roboterprogrammiersprachen anzugleichen. Insbesondere werden in dieser Arbeit Ansätze zum Umgang mit kontinuierlicher Veränderung, ereignisgesteuerten Aktionen und spezialisierten Basisprozessen in GOLOG beschrieben. Hierzu wird die Ontologie des Situationenkalküls um sich kontinuierlich verändernde Größen erweitert, was zum Beispiel eine adäquatere Modellierung der sich verändernden Roboterposition ermöglicht. Zur tatsächlichen Ausführung von Plänen wird eine geschichtete Kontrollarchitektur vorgeschlagen, in der eine Plan- und Kontrolleinheit mittels Nachrichten mit spezialisierten Basisprozessen kommuniziert. Der Formalismus erlaubt die Projektion von Plänen, ihre tatsächliche Ausführung, sowie das Spezifizieren von Plänen die zur Ausführungszeit Teilpläne projizieren und deren weitere Ausführung vom Ergebnis der Projektionen abhängt. Hierbei können auch probabilistische Unsicherheit sowie unsichere Sensoren und Effektoren berücksichtigt werden. Neben dem logischen Formalismus beschreibt die Arbeit eine prototypische PROLOG-Implementierung des Ansatzes sowie deren Evaluierung in verschiedenen Experimenten, unter anderem in einem Szenario, in dem der mobile Roboter CARL in der Abteilung Informatik V der RWTH Aachen als Bürobote agiert.

Contents

Chapter 1

Introduction

In the last five years, substantial progress has been made in building autonomous mobile robots which can navigate safely in populated areas like office environments, museums, or the like [KBM98, BCF+00]. We now have fairly robust solutions to basic "low-level" tasks like obstacle avoidance [FBT97] or self-localization [GBFK98, FBT99, FBDT99] so that it is possible to more seriously think about high-level control issues, that is, telling the robot what to do and how to do it. Intuitively, high-level control is concerned with the appropriate coordination of tasks like navigation in order to achieve the robot's overall goals.

Among the peculiarities of high-level robot control, we have the fact that the actions performed by the high-level controller are executed in order to achieve effects in the world outside the robot, and hence *outside the computer* controlling the robot. Similarly, goals pursued by the high-level controller and many objects it is dealing with lie outside the computer. For example, in a mail delivery application, the purpose of the high-level controller is to have letters – clearly objects lying outside the computer – moved to a new destination, like for example to the recipient's desk. Thus, unlike most programs which deal with objects inside the computer, a high-level robot controller is concerned with the effects of robot programs on the state of world. Another feature of high-level robot control is that autonomous robots have to robustly perform different and changing tasks in dynamic and incompletely known environments.

A central concept in the context of high-level robot control is the notion of a *plan*. Here, we adopt the view of McDermott [McD92a] who takes plans to be "that part of the robot's program whose future execution the robot reasons about explicitly". Given that the purpose of a high-level controller is to cause effects in the world, that is on objects outside the computer, this reasoning is (at least implicitly) concerned with the effects of the plan on the state of the world. The ability to do automated reasoning about the task at hand, in particular, the ability to *project* the outcome of a given plan or program, is important not only because it is an integral part of intelligent behavior such as rationally choosing among different courses of actions but also for pragmatic reasons. Note that projecting a plan can be thought of as a (qualitative) simulation of how the world evolves when actions are executed, which is quite helpful for debugging purposes. This is especially true for plans with concurrent actions, which arise naturally in robotics applications. Moreover, simulations are in general much faster than actually running tests on the robot.

Preferably, then, a high-level controller should make use of programs, or plans, which are not only suitable for (on-line) execution, but also for the purpose of projection, which is

sometimes also called off-line execution. However, as McDermott notes in [McD92a], there is a trade-off between the expressivity and "planability" of a plan notation, where planning is "explicit reasoning about future execution" of plans (or, emphasizing the purpose of planning, "the automatic generation, debugging, or optimization of robot plans").

At one end of the spectrum, we have the classical area of planning (e.g. [MR91, BF95, KS96]) in the tradition of Green [Gre69] and STRIPS [FN71], which aims at the automatic generation of a plan to bring about a desired state of affairs, given a description of the domain and of the effects of the robot's actions. When it comes to high-level control for robots in dynamic and incompletely known environments, however, these approaches suffer from at least two drawbacks: for one, the resulting plans are not adequate as a representation of real high-level robot programs. In their basic form, planners only consider sequences of primitive actions as possible plans, and this restriction is incompatible with the inherently concurrent nature of high-level robot control, which is due to the fact that sensors and effectors run in parallel.[1] For another, the approach to synthesize plans may end up being too demanding computationally in more complex settings (cf. [Byl94, LGM98]). The latter is particularly true for planners that take into account the robot's uncertainty (e.g [SW98]), planners that generate conditional plans that appeal to sensing (e.g. [WAS98, GA99]) or planners that account for probabilistic action effects (e.g. [DHW94, ML98]).

At another extreme, we have languages like RAP [Fir87], PRS [GI89, Mye96], RPL [McD91] and Colbert [Kon97].[2] These control languages provide concepts like concurrency, sensor-integration, interrupts, priorities, semaphores, and communication with lower-level routines for navigation or localization, to which we refer to as *low-level processes*. As a result, they allow very natural formulations of robot controllers. The drawback of these notations is that they do not rely on a formal framework for reasoning about actions, and thus that it is difficult to reason about whether a program is executable and whether it will satisfy the intended goals.[3] Among them, only RPL allows for plan projection. RPL's projection mechanism called XFRM [McD92a, McD94] is problematic, however, because projections rely on using RPL's run-time system, which lacks a formal semantics and which makes predictions implementation dependent.

Somewhere between these extremes, we have the action programming language GOLOG [LRL+97], which allows the projection of plans based on a perspicuous declarative semantics. GOLOG, which offers constructs such as sequences, iterations and recursive procedures to define complex actions, is unique in that its semantics is based on the situation calculus [McC63, LPR98], a logical language for reasoning about dynamic worlds. Roughly, the ontology of the situation calculus is based on *situations*, describing possible states of the world, and *actions* which cause the world to evolve from one situation to a successor situation. In

[1] Although many planners produce *partial-order plans* which do not prescribe a total order on the actions of the plan (e.g. [MR91, PW92]), all that is usually meant by the partial order is that any compatible order is allowed.

[2] One might object that languages like RAP or PRS are merely robot programming languages. In fact, in [McD91] McDermott refers to RPL and its relatives as a "family of notations for writing reactive plans for agents (e.g. robots)." Although some of these languages really are at the borderline of being plan languages, they provide at least a flavor of the ingredients necessary for reasoning. For example, as Firby states in his Ph.D. thesis [Fir89], "RAPs are designed to be used in two different ways: as abstract planning operators, and as programs to be run by the RAP interpreter".

[3] We remark that several proposals have been made to combine the use of a deliberative "classical" planner on top of a *sequencer* layer running a reactive program written in PRS, RAP or the like within a layered control architecture.

GOLOG programs, all primitive actions are the actions of the underlying situation calculus theory. Similarly, tests and conditionals in GOLOG do not refer to ordinary variables in the robot's computer memory, but to properties of the world (or, more exactly, to fluents[4] in the situation calculus axiomatization of the world). As a benefit, GOLOG supports projections firmly grounded in logic.

On the downside, however, GOLOG lacks the expressiveness provided by non-logic-based robot control languages like RPL, and is not expressive enough for realistic robot domains, first successful experiments using GOLOG to control a real museum-tour-guide robot [BCF+00] notwithstanding. Although there are extensions of GOLOG, in particular the concurrent programming language ConGolog [dGLL97, dGLL00] which provides additional facilities like prioritized execution of concurrent processes and interrupts, these still do not go far enough. Among the things that are missing we have at least the following:

1. *Dealing with Continuous Change*

   In the situation calculus (which underlies GOLOG), the world changes in a discrete fashion. However, in the context of mobile robots many changes are best thought of as continuous. For example, while moving, the robot changes its position continuously. The same holds for the battery level or the passage of time. While it may be possible to approximate such changes by discrete approximations, this seems at least unnatural and often adds considerable complexity to the reasoning involved.

2. *Time and Event-Driven Behavior*

   In the current temporal extension of GOLOG [Rei98], the user has to explicitly supply the time of execution for each action. However, when specifying a robot's task, this seems rarely appropriate and is often infeasible, especially in the context of concurrency. For example, suppose we want to tell the robot to do the following: (1) deliver today's mail to the offices; (2) whenever you pass near Gerhard's room say "hello"; (3) whenever the battery level drops dangerously low, interrupt whatever you are currently doing and re-charge your batteries. Notice that nowhere do we say explicitly when an action has to be taken. Instead, actions are conditioned on certain events happening like passing a certain office or reaching a low battery level. We call this event-driven behavior.

3. *A Layered Control Architecture*

   In order to pursue its goals, the high-level controller has to interact with specialized routines which we call *low-level processes*, and which provide basic-level capabilities like navigation or grasping objects. While in the original GOLOG framework all "actions" are represented as atomic situation calculus actions, typical "robot actions" like a low-level navigation process take time, and thus can not adequately be represented by an atomic, duration-less action. While there are proposals to represent such "robot actions" by means of a *pair* of duration-less actions representing their *initiation* and *termination* [Pin94, Ter94, Rei96] these approaches do not go far enough. In particular, they do not distinguish the nature of initiation and termination events, but uniformly treat them as actions executed by the high-level controller (cf. [Rei98]). However, while the initiation of a navigation process is under control of the high-level controller, its termination is an *exogenous* action, that is an action not under the control of the high-level controller.

[4] A fluent is a predicate whose truth value depends on the situation under consideration; see Section 3.1.

While this means that during actual execution the high-level controller can merely wait for the termination of the (navigation) process, for the task of projection it needs a model of the behavior of the low-level process, which provides an estimate when the low-level process will complete execution.

4. *Reasoning about Probabilistic Effects and Noisy Sensors*

   Actions in the situation calculus always have deterministic effects, that is, there is no uncertainty about whether or not an action achieves the desired results. As a result, the language GOLOG, whose tests and primitive actions refer to the underlying situation calculus model, do not allow for probabilistic reasoning.

   In practice, however, uncertainty seems to be ubiquitous, which is aggravated by the shortcomings of today's robots. Consider, for example, a pickup action. Given a certain characteristic of the gripper and the object to be lifted, we may want to say that the pickup action succeeds 80% of the time and fails otherwise, which, in its simplest form, may amount to having no effect at all. Similarly, sensor informations gathered by the high-level controller are subject to noise. For example, we may want to model that a letter recognition process has a 10% probability to overlook a letter to be delivered. Given such a probabilistic characterization, the high-level controller should not only be able to project the probabilistic effects of candidate plans, but should also update its beliefs during actual on-line execution (in particular, take into account noisy sensor information gathered). These two reasoning tasks are different in nature because the former does not involve any actual action or sensing information, while the latter deals with actual on-line execution.

1.1 Goals and Contributions

The goal of this thesis is to extend the GOLOG framework to represent, reason about and execute concurrent, event-driven plans in dynamic domains involving continuous change and probabilistic effects, thus shortening the gap in expressiveness between non-logic-based and logic-based robot control languages. The work reported here thus fits into the research area of Cognitive Robotics [LR98], which is concerned with the theory and the implementation of robots that reason, act and perceive in changing, incompletely known, unpredictable environments.

The framework presented in this thesis allows the specification of concurrent, event-driven plans. It supports the projection of candidate plans firmly grounded in logic, based on a precise representation of the interaction of the high-level controller with low-level processes and a model of the effects of the low-level processes on the world. The framework accounts for time, continuous change, probabilistic outcomes and noisy sensing. Although the automatic generation of plans within this setting becomes infeasible (at least computationally), the resulting framework helps the user in developing and experimenting high-level controllers. Not only does it provide the user with the possibility to generate projections of plans during development, but it also allows the specification of plans which appeal to the robot's beliefs at execution time and to on-the-fly projections of sub-plans, resulting in an interleaving of projection and on-line execution. The latter means that a plan may condition the execution of actions on the projected effects of a sub-plan, which allows the programmer to provide domain dependent procedural knowledge in a natural way. As an example, it is possible to execute a

sub-plan only if it is projected to have a reasonable probability to achieve a goal.[5] Besides, plans can be executed on-line, and we show how the resulting high-level control framework can be coupled to a real robot system, namely to the BeeSoft basic-task execution system which has successfully been used to control the mobile robots RHINO [BBC+95, BCF+00] and MINERVA [TBB+99].

In particular, this thesis makes the following contributions:

Dealing with Continuous Change, Time and Event-Driven Actions In robotics applications, we are faced with processes such as navigation which cause properties like the robot's location and orientation to change continuously over time. In order to model such processes in the situation calculus in a natural way, we add continuous change and time directly to its ontology, based on previous work by Pinto and Reiter [Pin94, Rei96, Rei98]. Furthermore, we adapt the semantics of ConGolog to the resulting temporal situation calculus. In particular, the resulting dialect of ConGolog, which we call cc-Golog, provides a new instruction *waitFor*(τ) which allows the plan to wait until a condition τ whose truth depends on the value of continuously changing properties becomes true. Using *waitFor*, it is possible to specify *event-driven* actions, that is actions whose execution time depends on certain conditions becoming true. For example, this allows a quite natural specification of tasks like "say hello if you come near Door 6213" by using an expression τ that verifies that the robot's location is near Door 6213 within a *waitFor* instruction.

A Layered Control Architecture As mentioned above, the initiation action that activates a low-level process and the termination action that represents its completion process are of different nature. In particular, while the activation action is under control of the high-level controller, the termination action is not. Actually, during on-line execution the termination action is exogenous, and during projection is has to be *simulated* by a model of the low-level process. In order to correctly account for both modes of execution (i.e. on-line execution and projection, which is also called off-line execution), we show how a robot control architecture where a high-level controller communicates with low-level processes via messages can be modeled directly in our situation calculus framework. The main advantage is that there is a clear separation of the actions of the high-level controller from those of low-level processes like a navigation process. Thereafter, we show how cc-Golog can be used to specify a *simulated environment* for the purpose of projection. For example, a navigation process can be modeled by a cc-Golog program which specifies that some seconds after its activation the navigation process will produce an action that signals that the destination has been reached.

On-Line Execution of cc-Golog Plans While during projection the time point of a *waitFor*-condition like a low battery level is computed based on an idealized *model* of the world, during actual execution, of course, the robot should react at the *actual* time where a condition becomes true. For example, the robot should react to the *actual* battery level by periodically reading its voltage meter. In order to use the same cc-Golog program both for online execution and projection, we explicitly distinguish between both modes of operation in

[5] The idea to interleave on-line execution and projection is due to [dGL99b]. However, while in their approach projection is used by the interpreter to resolve nondeterministic specifications, in our framework projection during on-line execution is explicitly required in test and branch conditions.

the semantics of *waitFor* instructions. In particular, we treat *waitFor*'s simply as special tests during online execution, and make use of frequent exogenous actions to provide the high-level controller with the latest estimates of continuous properties like the robot's position.

Interleaving On-Line Execution and Projection under User Control Motivated by the work on incremental execution of [dGL99b], which suggest that a combination between online execution and projection (which they call off-line execution) arises as a practical and still powerful scheme of execution, we add a local lookahead constructor which allows projection under user control. In particular, a plan may condition the execution of actions on the *projected outcomes* of a sub-plan. This can be done by means of branch instructions whose branch condition appeal to the result of a projection. For reasons of efficiency, we even go beyond that and allow for a restricted projection of a program, which only searches up to a (temporally) limited horizon. This allows the programmer to provide domain dependent procedural knowledge in a natural way. For instance, the programmer can specify that if the robot is in the middle of a delivery but near the docking station, then it has to make use of time bounded projection to find out whether the coming activities would allow it to operate for at least another 5 minutes and, if not, charge its batteries first.

Representing Noisy Low-Level Processes In realistic robot scenarios, the high-level controller is inherently uncertain in what the world is like and the outcome of many of the robot's low-level processes, due to the fact that robot hard- and software is imperfect and error-prone. For example, a pickup process may only succeeds 80% of the time and fail otherwise. In order to represent low-level processes with probabilistic outcomes, we model them as probabilistic programs in a probabilistic extension of GOLOG which we call pGOLOG. The intuition is that the different probabilistic branches of the programs correspond to different possible outcomes of the processes. For example, we can model the noisy pickup process mentioned above by a program which results in the pickup being successful with probability 80%, and else remains effect-less. Given a faithful characterization of the low-level processes in terms of pGOLOG programs, we can then reason about the effects of their activation through simulation of the corresponding pGOLOG models.

Representing Noisy Sensor Processes In our layered control architecture, sensing is handled by means of low-level processes like a door-state estimation process which, soon after their activation, provide information about the state of the world. We call such low-level processes *sensor processes*. For example, once a door state estimation processes has been activated, it makes use of physical sensors and appropriate pre-processing algorithms to estimate the opening angle of the door, and finally issues a message to provide the high-level controller with its estimate.

Thus, to us sensing means: activate a *sensor process*, which, in turn, has as effect a "sensor reading." Roughly, we represent the "answer" of a sensor process by means of exogenous actions, which results in a view of sensing which significantly differs from the well-known sensing actions of Scherl and Levesque [SL93, Lev96]. While during on-line execution the actual low-level process provides the answer, for the task of projection we model the behavior of the sensor process by means of a probabilistic pGOLOG program, which includes the very same exogenous actions as those used to represent the "answer" of the sensor process. Thereby, we represent the noisy outcomes of the sensor process by different probabilistic branches in the

pGOLOG program, which execute different exogenous actions.

Probabilistic Projection Based on a pGOLOG model of the low-level processes and a probabilistic characterization of the robot's epistemic state due to [BHL95], we show how to project high-level plans, appealing to a probabilistic model of the world and the low-level processes. As projections now yield the *probability* of a plan to achieve a goal, this leads us to the notion of *probabilistic projection*.

Belief Update in pGOLOG Besides being able to generate probabilistic projections of candidate plans, a high-level controller should update its beliefs during actual on-line execution, for example take into account noisy sensor information gathered. In particular, the ability to update the probabilistic belief state according to the occurrence of actions is required to allow the projection of plans in non-initial situations. This is quite important to accomplish intelligent behavior over an extended period of time, because after the execution of a first plan we want the robot to achieve further goals, which necessitates the projection of new candidate plans based on the updated belief state.

Based on pGOLOG programs which model the possible outcomes of all low-level processes (in particular of sensor processes), we show how probabilistic belief update can be realized within the pGOLOG framework. Roughly, after the activation of a low-level process, the beliefs are updated by assuming that any probabilistic outcome of the program can have happened. On the other hand, if a sensor process provides an answer, the pGOLOG model of the sensor process is used to sharpen the robots belief state in a Bayesian manner [RN95].

Implementation and Coupling to a Real Robot System Besides the formal specifications, we sketch a possible implementation of an interpreter for both cc-Golog and pGOLOG in PROLOG. As in the case of ConGolog [dGLL00], the implementation follows quite naturally from the formal specification of cc-Golog respectively pGOLOG, which consists of predicate calculus axioms. However, our implementation differs from the theory in it makes the usual *closed world assumption* on the initial database and makes use of PROLOG set-predicates like `findall` [SS94] where the theory appeals to second-order logic. Finally, we describe how the PROLOG implementation can be coupled with the BeeSoft execution system, a basic-task execution system which was successfully used to control the museum tour-guide robots RHINO [BBC+95, BCF+00] and MINERVA [TBB+99].

1.2 Outline of this Thesis

The rest of this thesis is organized as follows.

- In Chapter 2 we discuss related work in the areas of reasoning about action and change, cognitive robotics and non-logic-based robot control.
- In Chapter 3 we give a formal introduction to the situation calculus, the logical framework on which GOLOG and its derivatives is based on. Thereafter, we introduce the language ConGolog and give a formal semantics for it. Finally, we describe a methodology due to Bacchus, Halpern and Levesque [BHL95] which allows the representation of the robot's probabilistic belief state within the situation calculus.

- In the Chapter 4, we show how to extend the situation calculus to include continuous change and time. Thereafter, we present cc-Golog, a derivative of ConGolog which takes into account the extended situation calculus, in particular its notion of time and temporal precedence. Finally, we show how a robot control architecture where a high-level controller communicates with low-level processes via messages can be modeled directly in cc-Golog, and show how cc-Golog can be used to provide natural specifications of event-driven tasks like immediately interrupting a delivery to charge batteries when the battery level drops dangerously low.

- In Chapter 5, we discuss the changes needed to allow both on-line execution and projection of cc-Golog programs, and to allow the user to interleave on-line execution and projection.

- In Chapter 6 we introduce the probabilistic language pGOLOG. Thereafter, we model the robot's probabilistic epistemic state following [BHL95, BHL99], and show how noisy low-level processes, in particular sensor processes, can be modeled using pGOLOG. Finally, we show how probabilistic projection works in pGOLOG.

- In Chapter 7, we show how belief update can be achieved within the pGOLOG framework. In particular, we show how the robots confidence decreases when the robot activates noisy low-level processes, and how it is sharpened when a sensor process provides information. Thereafter, we show how our framework can be used to replicate an example considered in [BHL99]. Finally, we show how to interleave online execution and probabilistic projection.

- In Chapter 8 we sketch the implementation of a cc-Golog and a pGOLOG interpreter in PROLOG, and provide some experimental results regarding the runtime of our prototype implementation. This is followed by a description of an execution system which couples cc-Golog respectively pGOLOG to a real robot, namely to the BeeSoft execution system [BBC$^+$95, BCF$^+$00].

- Finally, we end with a summary and concluding remarks in Chapter 9.

Some of the results reported in this thesis have already been published before. In particular, the language cc-Golog described in Chapter 4 was introduced in [GL00a]. The approach to on-line execution of cc-Golog plans presented in Chapter 5 was first proposed in [GL01b]. The probabilistic language pGOLOG described in Chapter 6 was introduced in [GL00b], and finally the approach to belief update presented in Chapter 7 is based on earlier work reported in [Gro00, GL01a].

Chapter 2

Related Work

In this chapter, we briefly summarize the most relevant existing work related to our approach to high-level robot control. The chapter is subdivided into three sections. In the first section, we go over some of the most important formalisms for representing and reasoning about action and change. The next section introduces the high-level programming language GOLOG [LRL+97] and discusses its various derivatives. Finally, in the last section we consider the area of non-logic-based robot control, elaborating on the robot programming language RPL whose expressive power has largely motivated the approach presented in this thesis.

2.1 Reasoning about Action and Change

As described in the previous chapter, the purpose of a high-level controller is to cause effects in the world. Hence, a substantial task in reasoning about the effects of a plan is to reason about its effects on the state of the world, and in particular to reason about the effects of the execution of primitive actions on the state of the world. In this section, we will provide a brief survey of the most important approaches to reasoning about actions and their effects, also known as *reasoning about actions and change*. We will focus on logic-based approaches, in particular on a formalism called the *situation calculus*.

2.1.1 The Situation Calculus

The situation calculus [McC63] is a logical language designed for representing dynamically changing worlds. It characterizes the world as consisting of a sequence of *situations*, which can be understood as "snapshots" of the state of the world. All situations are first-order citizens of the logical language. In particular, there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred. All changes are the result of *primitive actions*. Just like situations, actions are first order terms and are denoted by functional symbols. For example, the term $gotoRoom(R_2)$ could be used to represent the action of the robot moving to room R_2. All actions have *preconditions*, requirements that must be satisfied for the action to be executed. For example, a robot can only give someone a letter if it has loaded that letter.

To represent how the world changes from one situation to its successor, the situation calculus includes a distinguished binary function symbol do, where $do(a, s)$ denotes the situation that results from performing action a in situation s. For example, $do(gotoRoom(R_2), S_0)$

would represent the situation where the robot has moved to room R_2, and nothing has happened before. In order to represent time-varying properties of the world, like for example the robot's location, the situation calculus offers the concept of a *fluent*, a relation or function which takes a situation as last argument. For example, the function symbol $robotLoc(s)$ could be used to represent the location of the robot in a particular situation s. "Causal laws" specifying how primitive actions affect the value of fluents are specified by *effect axioms*. For example, an effect axiom could specify that after the execution of a $gotoRoom(R_2)$ action the value of *robotLoc* is R_2.

The Frame-, the Qualification- and the Ramification-Problem Besides specifying which facts change as a result of the execution of an action, one needs axioms specifying which facts remain unaffected by the action. For example, one needs axioms stating that a *gotoRoom* action does not change the color of a letter loaded. The "non-effects" of actions are described by *frame axioms*. Together, effect axioms and frame axioms provide a complete description of how the world evolves in response to the agent's actions. One of the early problems encountered in using the situation calculus (as well as other formalisms for reasoning about actions) is known as the *frame problem*: it arises because in general the number of frame axioms is very large, which complicates the axiomatizing a domain and can make theorem proving extremely inefficient. In the next Chapter, where we will provide a more formal introduction to the situation calculus (Section 3.1), we will describe a simple solution to the frame problem due to Ray Reiter where *successor state axioms* are used as a compact representation of both effect and frame axioms [Rei91].

At least two other problems have surfaced in the study of reasoning about actions: the *qualification* and the *ramification problem*. The qualification problem arises because in the real world it is difficult to explicitly specify *all* the necessary and sufficient conditions for an action a to be executable. For example, a robot can obviously only hand over a letter if it has actually loaded this letter, but even so the robot's gripper may be damaged, the letter may be glued to the gripper etc. The ramification problem has to do with implicit consequences of actions. For example, if the robot is covered with dust or bacteria, then while moving around the robot also carries each particle that adheres to the robot. Though it might be possible to explicitly describe such effects as part of the description of a *gotoRoom* action, one would prefer to describe such indirect effects in a modular way. A large body of work exists dealing with these two issues (e.g. [LR94, Thi01b]). However, in this thesis we are mainly concerned with other issues, so we will simply ignore these two problems.

In the remainder of this section, we will describe some extensions of the situation calculus which allow a more natural specification of real-world domains. In particular, we consider approaches to incorporate continuous time and change, knowledge and sensing, and probabilistic reasoning into the situation calculus, as these are important features of typical robotics applications.

Continuous Time and Change in the Situation Calculus In its basic form, the situation calculus has no quantitative notion of time - there is a partial order on situations induced by the function $do(a, s)$ but otherwise the different situations are not dated. In order to represent quantitative time in the situation calculus, Pinto and Reiter [PR93, PR95] introduce a functional fluent $start(s)$, which represents the starting time of situation s. The structure of time is considered to be isomorphic to that of the positive real numbers, and

each occurring action is associated with a time point at which the action is said to occur. To represent an action that has a duration (also called a process), they propose to make use of two instantaneous actions that represent the initiation and the termination of the process. The idea is to represent a process like picking up a block by a *start_pickup* and an *end_pickup* action.

In real-world applications, one is also often concerned with processes which cause properties to change *continuously* over time. For example, in a mobile robot application the robot's location and orientation change continuously as it navigates through its environment. To represent continuous change in the situation calculus, Pinto [Pin94, Pin97] proposes to represent the properties of the world using two different concepts: discrete fluents, and *continuous parameters*, which represent continuously varying properties of the world. A parameter can be stated to behave as described by a named *function of time* through the special binary fluent $\doteq$, which takes as arguments a parameter and a named function of time. The idea is to express, for example, that a ball is moving with velocity v by writing axioms of the form $\mathit{ballPosition}(s) \doteq \mathit{linear}(v, x_0)$, where $\mathit{linear}(v, x_0)$ is an (appropriately axiomatized) named function of time.

In [Rei96], Reiter proposes a version of the situation calculus where all primitive actions take as last argument a parameter denoting their occurrence time. The approach also considers truly *concurrent* actions which consist of a set of primitive actions, and *natural* actions which occur in response to known laws of physics. In particular, the approach makes use of action precondition axioms which determine the occurrence time of an action by means of numerical equations. For example, a *bounce* action representing that a falling ball bounces of the floor is specified to be possible only at the time where its height becomes zero, where the height of the ball at time t is calculated using the laws of physics. Natural actions are required to occur at their predicted times, provided no earlier action prevent them from occurring. For example, a *bounce* action will happen unless the ball is caught before.

Based on Reiter's framework, Kelley [Kel96b, Kel96a] demonstrates how natural actions can be used to model and reason about complex dynamic systems within the situation calculus.

Knowledge and Sensing in the Situation Calculus The approaches described so far were only concerned about representing and reasoning about the *actual* state of the world. However, in many real-world applications a robot or agent has only limited *knowledge* about its environment. As an example, consider a mobile robot whose goal is to get into a room. Furthermore, assume that the robot can travel into the room through one of two doors, and that it knows that one door is open, but that it does not know *which* door is open. Clearly, in this example there is no sequence of primitive actions which can be shown to achieve the goal.

In such applications, representing the limited knowledge of the agent and its ability to *sense* properties of the world is an important issue. For example, what is needed in our robot application is a sensing action which allows the robot to determine whether a specific door is open. Building on the work of Moore [Moo85], Scherl and Levesque provide a theory of knowledge and sensing in the situation calculus [SL93]. Briefly, they propose to use a new fluent K whose first argument is also a situation: informally, $K(s', s)$ holds when a robot in situation s, unsure of what situation it is in, thinks it could be in situation s'. Knowledge for the robot, then, is defined as what holds in *all* of these so-called *accessible*

situations. Furthermore, they introduce *sensing actions* into the situation calculus, actions which provide the robot with information about the state of the world. In particular, they consider binary sensing actions that tell the robot whether or not some condition holds in the current situation. Their approach is based on the use of so-called *sensed fluent axioms* which specify that a sensing action a can be used to determine the truth value of a condition ϕ_a. Based on these axioms, they formalize how the robot's knowledge - characterized by a set of situations considered possible - evolves from one situation to another. As an example, within this framework it would be possible to formalize that after execution of a *senseDoor* action, representing that the robot senses the state of a door, it *knows* whether or not the door is open.

Finally, in [Lak96, LL98], Lakemeyer and Levesque consider the issue of *only knowing* in the situation calculus, which is concerned with expressing that an agent knows a sentence, and *nothing* more. The idea is that if an agent *only knows* ϕ, then it considers *every* situation s' as possible which satisfies ϕ and hence has minimal knowledge except for the knowledge that ϕ holds.

Probabilistic Reasoning in the Situation Calculus Although the approaches discussed in the previous section account for uncertainty about the state of the world, they do so in a qualitative way - for example, the robot knows whether a door is open, or not. In real-world applications, however, an agent often has *degrees* of confidence in various propositions, and must cope with *noisy* sensors that can only be used to increase its beliefs in certain propositions. An appealing approach to deal quantitatively with this kind of uncertainty in the situation calculus is to integrate concepts from *probability theory*.

In [BHL95, BHL99], Bacchus, Halpern and Levesque (BHL) propose a framework for reasoning about noisy sensors and effectors in the situation calculus which is based on a new functional epistemic fluent p. Roughly, p can be thought of as a weighted version of Scherl and Levesque's epistemic fluent K [SL93]; that is, it can be read as "in situation s, the agent thinks that s' is possible with degree of likelihood $p(s', s)$." Based on p, they define $Bel(\phi, s)$, the agent's degree of belief that ϕ holds in situation s, as the (normalized) sum of the degrees of likelihood of all situations s' considered possible in s that fulfill ϕ. To represent that noisy sensors and effectors can have different possible outcomes, they make use of *nondeterministic* instructions, i.e. instructions which can result in many different outcomes (see also Section 3.2). For example, in an example domain where a mobile robot is moving along a straight line in a 1-dimensional world, they use the nondeterministic instruction $noisyAdv(x)$ which is specified to stand for *any* $exactAdv(x, y)$ action. Here, $exactAdv(x, y)$ stands for the robot trying to advance by x units but actually moving y units. The idea is that the robot cannot directly execute the primitive action $exactAdv(x, y)$, but instead can only execute the nondeterministic instruction $noisyAdv(x)$.

The main topic of their approach is to represent how the robot's beliefs change when it activates noisy sensors and effectors. In order to represent the agent's limited perceptual capabilities, they introduce *observation-indistinguishability axioms* (OI-axioms) which are used to specify that (in a certain situation) the agent is unable to discriminate a set of actions a' from the action a. For example, OI axioms are used to specify that the robot cannot distinguish two $exactAdv(x, y)$ and $exactAdv(x, y')$ actions. Additionally, *action-likelihood axioms* are introduced to specify the likelihood of an action a in situation s, for example to model that the agent believes that there is a 50% chance that $noisyAdv(x)$ will move the

robot by the exact distance x. Noisy sensors can be represented similarly. Based on these concepts, BHL define a successor state axiom for the epistemic fluent p, thereby formalizing how the robot's beliefs change as it activates noisy sensors and effectors. They illustrate that their framework correctly captures the intuitive effects of noisy sensors and effectors on the robot's beliefs, and show (under a few assumptions) that the robot's beliefs are updated in a manner identical to standard Bayesian conditioning.[1]

In [PSSM00], Pinto *et al.* propose a different approach to integrate probability theory into the situation calculus. Their approach is based on the idea that an agent cannot directly execute a primitive action, but instead can only provide an *input* which results in a transition, and that thereafter nature steps in and probabilistically chooses an actual *outcome*. Any primitive action in their framework thus consists of two components: the agent's input and the actual outcome. As an example, in a coin-toss example there is only one possible input, *toss*, and the possible outcomes (chosen by nature) are *heads* and *tails*. The primitive actions in this example thus are the pairs $\langle toss, heads\rangle$ and $\langle toss, tails\rangle$. As these primitive actions have deterministic effects, the resulting formalism can employ Reiter's solution to the frame problem.

The different possible reactions of nature are assigned probabilities. Based upon these probabilities, the probability that a fluent F holds after a sequence of inputs is defined in terms of the (deterministic) action sequences that can result from the inputs and nature's reactions. While the original framework is restricted to discrete probability distributions, in [MPP$^+$01] it is extended to domains where the set of possible outcomes is continuous. Basically, this is realized by relying on an "oracle" that provides the result of operations on real numbers, probability distributions and the like (in the implementation, the MATHEMATICA software system [Wol96] is used as oracle). Unlike the framework of BHL, this approach is not "agent centered," meaning that it does not consider the agent's beliefs, but only the external probability of success of possible actions. In particular, the approach does not consider sensing.

2.1.2 Other Approaches to Reasoning about Action and Change

While the situation calculus is one formalism for reasoning about action and change, there are many others. In this subsection, we will briefly go over the most important ones, focusing on logic-based approaches.

STRIPS The STRIPS planning system [FN71] was designed as the planning component for the (second version) of the robot Shakey [Nil84]. Its purpose was to generate a plan of actions that Shakey could then execute step by step in order to achieve a given goal. STRIPS uses a restricted, propositional language for representing the effects of actions, which lends itself to efficient planning algorithms. In the following, we use the term STRIPS to refer to STRIPS' representation language. Its central concept is the STRIPS *operator*, which is used to represent the available actions. A STRIPS operator essentially consists of a set of atoms which represent the *preconditions* that must be true before the operator can be applied, and a set of literals (positive or negative atoms) that describe its *effects*, i.e. how the situation changes when the operator is applied. The frame problem is solved by assuming inertia

[1] Roughly, Bayes' rule says that the probability of Y given evidence X corresponds to the product of the probability of X given Y and the prior probability of Y, divided by a normalizing factor; cf. [RN95].

regarding those propositions not affected by the operator (cf. [GL98a], where a transitions-based semantics for STRIPS is considered). A formal account of STRIPS which discusses its relation to the situation calculus can be found in [LR95].

While the early STRIPS-based planners where searching for plans consisting of a totally ordered set of operators, by now it is common to consider partially ordered plans (e.g. [Sac75, MR91]). A plan is partially ordered if some steps are ordered with respect to each other and other steps are unordered. A partially ordered plans is considered a solution to a planning problem if any linearization of it is a solution. The ADL formalism extends the (quite restrictive) STRIPS representation to allow for conditional effects [Ped89, PW92]. The area of *conditional planning* (e.g. [War76, PS92, WAS98]) aims at synthesizing plans involving sensing actions and conditionals. Probabilistic planners (e.g. [KHW95, ML98]) use a probabilistic extension of STRIPS where the operators can have different probabilistic effects. Probabilistic planners that accounts for sensing (e.g. [GA99, BL99]) additionally consider (noisy) information-producing actions, and consider plans whose steps can be conditioned on the outcome of (previously executed) information-producing actions. Several planning system consider (continuous) resources such as time, money and energy (e.g. [Ver83, Koe98, HG01]). STRIPS-planning is still an area of active research. Many of today's most efficient planners are based on plan graphs [BF95, KNHD97], planning as satisfiability testing [KS92, KS96], or heuristic planning [BG99, DK01].

The Event Calculus The event calculus was originally introduced by Kowalski and Sergot as a logic programming formalism for representing events and their effects, especially in database applications [KS86, Kow92]. Here, we follow the dialect of the event calculus due to Shanahan, described in [Sha99]. Its ontology comprises events (also called actions), fluents and time points. Events initiate and terminate periods during which fluents hold. Here, fluents (just as events and time points) are first-class objects, unlike in the situation calculus. The event calculus includes predicates for saying what happens when ("happens(action,time)"), for describing the initial situation, for describing the effects of actions, and for saying what fluents hold at what times.

The frame problem is solved using circumscription [McC80], a form of non-monotonic reasoning which is based on the idea to minimize the extension of certain named predicates (in a nutshell, a logic or formalism is *non-monotonic* if adding axioms to a theory may non-monotonically change the set of sentences entailed). In particular, the extension of the predicate happens, representing the set of action occurrences, is minimized, as well as the extension of predicates representing the positive and negative effects of actions. The key to this solution is the splitting of the theory into different parts, which are circumscribed separately. We remark that unlike Reiter's solution to the frame problem, which essentially works by incrementing the axiom set with additional axioms (cf. Section 3.1.1) and therefore represents a *syntactic* method for reducing the intended model set, the event calculus' solution to the frame problem is semantic. In the case where the theories are restricted to Horn Clauses, the circumscriptions reduce to predicate completions [Cla78] and the non-monotonic theory can be "compiled" into a classical (monotonic) logical theory. A significant difference compared with the situation calculus is that in the event calculus planning has to be handled as abduction, not as deduction.

The formalism can handle actions with indirect and nondeterministic effects, compound actions (which can include standard programming constructs such as sequences, conditionals

and recursion) and concurrent actions. In [Sha90], Shanahan presents an extension of the event calculus which is able to represent continuous change. The approach is based on the notion of the *trajectory* of a continuously changing quantity, which is a path plotted against time through the corresponding quantity space. In [MS96], the framework is extended to allow descriptions using arbitrary systems of differential equations, based on a case study by Miller on continuous change in the situation calculus [Mil96]. In particular, the extended framework includes the concept of a derivative function.

In [Sha96], Shanahan provides a logical account of sensor data assimilation in which a model of the actual state of the world is constructed through an abductive process. The approach is developed using the example of a mobile robot with simple proximity sensors. Essentially, the abduction process hypothesizes the existence and locations of objects to explain the sensor data obtained. The theory builds on a representation of space as $\mathbb{R}^2$, and of objects as 2-dimensional regions. The approach also represents noisy effects (in particular motor noise) by nondeterminism. For that purpose, the robot's location (which is represented as changing continuously) is only constrained to be within a *circle of uncertainty* centered on the location where it would be if its motors weren't noisy. In [Sha97], the framework is restated based on the method of determining fluents, which allows it to deal with noisy sensors. Again, noise is considered as nondeterminism.

Sandewall's Approach In [San89a], Sandewall proposes to combine non-monotonic temporal logic with differential calculus. The formalism is based on the concept of *parameters*, continuously changing properties which are assumed to be piecewise continuous. Parameters (which correspond to fluents) have a real-number value at each point in time. The possible histories over time may contain an arbitrary number of time-points called *breakpoints*, where the parameters may have discontinuities. In the intervals between two breakpoints, the parameters are assumed to be continuous. An example where a breakpoint arises is when an object held is let go, which causes its vertical acceleration to change from 0 to -9.81 m/s^2.

While differential calculus is used to characterize the parameters and their continuous intervals between breakpoints, a non-monotonic temporal logic is used to characterize the behavior of the parameters around the breakpoints. The non-monotonicity is needed in order to formalize that parameters are continuous at break-points whenever that is consistent with the axioms. For example, this is needed to ensure that if one object is let go (which means that there is a break-point), other objects retain their positions. The semantics of the logic is based on the concept of preferred interpretations, which involve minimal changes at breakpoints. This concept is similar to the principle of chronological minimization introduced by Shoham [Sho87, Sho88].

While the original approach was limited to the case in which there are no agents or actions in the world, in [San89b, San94] the approach is extended to the case where actions may occur. In the presence of actions, it turns out that the conventional preferential semantics, also called chronological minimization of discontinuities (CMD), will include anomalous models. In order to overcome this problem, Sandewall proposes a semantics based on the concepts of *filtering* and *occlusion* (or *masking relations*). The idea is to separate the axioms describing the observations (or goals, in a planning problem) from the formulas representing all the other given information (like action statements) and to first determine by means of CMD the set of all possible developments in the world regardless of any observation. Thereafter, this set of interpretations is "filtered" with the given observation, instead of applying CMD to the union

of the observation and the other axioms. Finally, the idea of occlusion is to minimize only those discontinuities which are not caused by actions. Note that this is similar to a solution to the frame problem, but for a continuous domain. In [DL94], Doherty and Lukaszewicz shows how this preferential entailment can be characterized in terms of standard predicate calculus and circumscription.

The Action Language $\mathcal{A}$ The action language $\mathcal{A}$ [GL93] is a simple (propositional) language for describing actions. The ontology of $\mathcal{A}$ is based on fluents and actions. A state is a set of fluents. A domain description consists of value propositions and effect propositions. Value propositions provide information about the value of fluents in the initial state or after execution of some actions. Effect propositions describe the effects of actions on the fluents. The semantics of $\mathcal{A}$ is based on the concept of a *transition function*, which is a mapping from a state and an action to a successor state. The entailment relation of $\mathcal{A}$ is non-monotonic in the sense that adding effect propositions to a domain description may non-monotonically change the set of propositions entailed.

In [SB98], Son and Baral include the notion of knowledge into the action language $\mathcal{A}$. Their approach is based on that of Scherl and Levesque [SL93]. Essentially, they extend the notion of a state, which in $\mathcal{A}$ is a set of fluents, to a k-state, which consists of a pair $\langle s, \Sigma \rangle$. Here, s is a state (representing the real state of the world) and Σ a set of possible states which the agent believes it might be in. So-called *k-propositions* specify which actions can be used to determine the truth value of a fluent. They also define conditional plans, which execute depending on the knowledge of the agent.

In [BT01], Baral and Tuan propose a probabilistic extension of the language $\mathcal{A}$. Basically, they add a new set to the language, the unknown variables. As in $\mathcal{A}$, there is also a set of propositional fluent symbols and a set of action symbols. Unknown variables can only occur in the antecedent of effect propositions, meaning that causal rules never affect the truth value of unknown variables. On the other hand, unknown variables can affect the value of fluents in a state. The underlying idea is that nature's laws are deterministic, and randomness surfaces due to the reasoner's ignorance of the underlying boundary conditions. Besides the domain specification, a domain description in probabilistic $\mathcal{A}$ includes a set of propositions which specifies the probability of the unknown variables. The formalism also includes an observation language which allows the user to state that certain facts hold initially or after the execution of some actions. Given a domain description, a specification of the unconditional probabilities and some observations, the user can then ask for the probability that a formula holds after the execution of some actions.

Causal Theories and the Language $\mathcal{C}$ The causal theories proposed in [MT97] are based on the principle of universal causation: every fact that obtains is caused. That is, they distinguish between the claim that a proposition is true and the stronger claim that there is a cause for it to be true. Roughly, a causal theory D is a set of causal laws of the form $\phi \Rightarrow \psi$. The idea is that a world history is (causally) possible if every fact that holds in it is caused, or put another way, that exactly the fact that obtain in it are caused in it. The formal semantics is based on a fixed-point definition, which states that, intuitively, I is a causally explained interpretation of a causal theory D if is is a model of its consequences with respect to the causal laws of D. The entailment relation of causal theories is non-monotonic. Based on this framework, it is possible to formalize domains which account for nondeterministic actions and

for facts which are explained by inertia, ramification and qualification constraints.

While the original approach only accounted for propositional causal rules, in [Lif97] the framework is extended to arbitrary non-propositional rules. In addition, the principle of universal causation is weakened in that it only applies to a subset of the symbols, the "explainable" symbols. This is achieved by a translation into second-order predicate logic. This extended framework is the foundation for the action language $\mathcal{C}$ [GL98b, GL98a]. In $\mathcal{C}$, there are two kinds of proposition: static laws and dynamic laws. As in $\mathcal{A}$, the semantics is based on transitions. Here, however, a transition is required to be causally explained. Only those fact hold after execution of an action which have been caused. Thus, inertia is not a built-in feature in the semantics of $\mathcal{C}$, unlike in $\mathcal{A}$. Roughly, in the semantics of $\mathcal{C}$, the principle of universal causation is applied to the world histories that consist of two time instants - before and after executing an action - and restricted to the facts obtained in the second of these time instants.

The Fluent Calculus The aim of the fluent calculus [Thi99c, Thi99b] is to provide a more efficient way to compute the (non)-effects of actions (this is sometimes called the "inferential" frame problem). One motivation for the fluent calculus is that in the situation calculus, in order to solve a projection task one has to carry the truth value of any fluent from the point of its appearance past each intermediate situation to the point of its use. This has to be done one-by-one, and using separate instances of the relevant axioms.

The idea behind the fluent calculus is that instead a single *state update axiom* will suffice to derive the entire change caused by the action in question. To achieve this, the fluent calculus extends the ontology of the situation calculus from which it descends in that it distinguishes between the concept of a situation (which is a sequence of actions), and a *state*. A state is simply a collection of fluents, which are reified to this end, that is treated as first-order objects. Each situation is related to a state, which characterizes the state of the world in that situation. State update axioms specify how the states at two consecutive situations are related, that is which fluents become true respectively false by the execution of a primitive action. Similar to Reiter's solution to the frame problem [Rei91], the concept of state update axioms is based on the assumption that a set of axioms is *complete* in the sense that it specifies *all* relevant effects of all actions involved [Thi98].

While the use of state update axioms allows to derive the entire change caused by actions using significantly fewer instances of axioms, it is not clear whether in practice the fluent calculus is always superior to the situation calculus because the use of state update axioms requires extended unique names axioms for states, which compute complete sets of most general unifiers with respect to the equational theory of associativity, commutativity, and existence of unit element. Besides, state update axioms require that an action does not have potentially infinitely many effects.

In [Thi99a], Thielscher shows how to integrate continuous change into the fluent calculus. [Thi00b] describes how Scherl and Levesque's framework for dealing with knowledge and sensing in the situation calculus [SL93] can be adapted to the fluent calculus with the additional benefit that the resulting framework also allows to distinguish between the actual effects of actions, and what a robot knows about these effects (for example, there might be a button which causes a door to open, but the robot does not know that the button is causally related to the door).

(PO)MDPs Another principled approach to reasoning and acting under probabilistic uncertainty is the theory of Markov decision processes [Bel57, Put94]. A Markov Decision Process (MDP) is defined in terms of the following components: a finite set of *states*; a finite set of *actions* which influence the system state; a *transition function* that, given a state and an action, returns a probability distribution over resulting states; and, finally, a *reward function* which specifies the reward given in each state. The process is fully observable in the sense that although the agent cannot predict the outcome of an action with certainty, it can observe the state once it is reached. The decision problem faced by an agent in an MDP is that of forming an optimal *policy*, that is a mapping from states to actions that maximizes the expected total accumulated reward over some (finite or infinite) horizon. A partially observable MDP, or a POMDP, is an MDP together with a set of observables and an observation function that specifies the probability distribution for the observables for each state and action [KLC98]. In a POMDP, the agent is unable to observe the current state. Instead, it makes an observation based on the action and resulting state. The agent's goal remains to maximize its total expected reward.

One difficulty faced by classical approaches to (PO)MDPs like dynamic programming [Bel57] is their reliance on an explicit state-space formulation, which causes their complexity to be exponential in the number of state variables. As a result, the computational cost rapidly becomes prohibitive already in relatively small domains (cf [GB98]). Recent work in decision theoretic planning has focused on the development of compact, natural representations for POMDP [BP96, GB98]. One possibility is to use Bayesian networks [Pea88], which provide a compact representation of the dependencies between probabilistic variables. For example, Dean and Kanazawa [DK89] use two-stage temporal Bayesian networks which represent the transition probability associated with an action by two disjunct sets of nodes. One set of nodes represent the state prior to the action, while another set represent the state after the action has been performed. [BRP01] consider first-order representations of state spaces and use an operation called *decision-theoretic regression* to compute policies.

Other Approaches Other approaches to reasoning about action and change include the temporal logics by McDermott and Allen [McD82, All84, AF94]. Herrmann and Thielscher [HT96] propose a calculus based on the notion of processes where a situation is characterized by a (real-valued) time point together with a set of processes, and where transition laws involve numerical equations specifying when processes end and new processes begin. Finally, the area of qualitative reasoning [Hay79, Hay85, For84, Kui86] is concerned with representing continuous changing properties using discrete qualitative representation.

2.2 GOLOG and its Derivatives

As described in the previous section, the situation calculus respectively its various extensions are well suited to represent and reason about the effects of primitive actions in complex robotics domains. The same holds for (some of) the other approaches described in Subsection 2.1.2, like the event calculus or the fluent calculus. However, in robotics applications we would not only like to reason about the effects of primitive actions, but about the effects of complex high-level robot *programs*. For that purpose, the use of a situation calculus based approach is suggestive because there exists a family of powerful high-level programming languages built on top of it.

GOLOG and ConGolog GOLOG [LRL+97] is a high-level programming language designed for the specification of complex behavior in dynamic domains, like for example high-level controllers operating mobile robots. GOLOG comes with a declarative semantics which is based on the situation calculus. In particular, every primitive GOLOG instruction is a primitive action of the underlying situation calculus theory. As a result, the GOLOG interpreter can automatically maintain a model of the world during the execution of a plan simply by keeping track of the primitive actions being executed. Similarly, it is straightforward to reason about the effects of GOLOG plans on the state of the world.

Besides executing primitive actions, a GOLOG program can test properties of the domain by appealing to the value of fluents, and similarly can condition the execution of actions on the value of fluents or complex formulas. GOLOG also provides instructions like sequences, iterations and recursive procedures to define complex programs. Additionally, GOLOG provides *nondeterministic* instructions. This allows the specification of nondeterministic programs like "execute action a *or* action b, and verify that afterwards ϕ holds," with the idea that the GOLOG interpreter shall automatically search for a sequence of actions that constitutes a legal execution of the nondeterministic program. In doing so, the GOLOG interpreter makes use of the underlying situation calculus theory to reason about the effects of various courses of actions. A prototype implementation of GOLOG has been developed and has already been used to control different real robots [LTJ98, BCF+00]. In particular, Hähnel *et al.* [HBL98] describe how GOLOG can be coupled to the quite successful BeeSoft robot control system [BCF+00].

In [dGLL97, dGLL00], instructions for concurrent execution are introduced with a conventional interleaving semantics. Besides this concurrent extension called ConGolog, several other extensions of GOLOG have been proposed. In the remainder of this section, we will briefly discuss the most important ones and sketch their underlying ideas.

Sequential, temporal GOLOG In [Rei98], Reiter proposes a temporal extension of GOLOG which is based on the temporal situation calculus presented in [Rei96]. In particular, each primitive action has a starting time, which has to be explicitly stated in the program. Actions with duration (processes) are represented by instantaneous actions that represent the initiation and the termination of the process. For example, to represent that a robot travels from a location loc_1 to a new location loc_2, in sequential, temporal GOLOG one would make use of a program like $[startGo(loc_1, loc_2, t); endGo(loc_1, loc_2, t + travelTime(loc_1, loc_2))]$, where the robot first starts moving from loc_1 to loc_2 at time t, and then stops moving at time $t + travelTime(loc_1, loc_2)$.

While this representation is well suited for the projection of a plan, it would be unrealistic to expect the robot to execute a program like the above and to meet the exact times in the resulting schedule of actions. One approach to overcome this problem is to make use of *execution monitoring*. Execution monitoring is the robot's process of observing the world for discrepancies between the actual world and its internal representation of it, and recovering from such discrepancies [dGRS98]. In the case of temporal programs, the execution monitor's (primary) task would be to adapt the schedule to the actions' actual occurrence time.

On-line Execution of GOLOG Programs The GOLOG dialects considered so far execute in an *off-line* manner: given a program, the interpreter finds a sequence of actions constituting an entire legal execution of the program before actually executing any of them in the world.

In [dGL99b], de Giacomo and Levesque argue that this execution mode is problematic when large nondeterministic programs are to be executed, and, maybe more fundamentally, in the presence of sensing actions. As an alternative to the off-line execution style, they propose to consider the *on-line* execution of programs. In the on-line execution of a GOLOG program, nondeterministic choices are treated like *random* ones, and any action selected is executed immediately.

The on-line style of execution is well-suited to programs containing binary sensing actions [SL93]. An off-line GOLOG interpreter cannot predict the outcome of a sensing action, and the best it can do is search for a sequence of actions that is a legal execution for both possible outcomes of a sensing action. On the other hand, an on-line interpreter does not have to reason about the impact of future sensing actions when executing a plan and can nevertheless take into account the outcome of sensing actions. Of course, on-line execution of programs involving a lot of nondeterminism is also much faster, because the random execution style does not require any reasoning. However, this execution style has the serious drawback that it does not guarantee that the execution will successfully complete. To get the best of both approaches, de Giacomo and Levesque propose to interleave on-line and off-line execution. In particular, they introduce a new operator Σ for off-line search, so that $\Sigma\delta$, where δ is any program, means "consider δ off-line, searching for a globally successful termination state."

sGolog Another approach for dealing with sensing actions is Lakemeyer's sGOLOG [Lak99]. Unlike the approach of de Giacomo and Levesque, sGOLOG is based on off-line execution. One motivation is that the on-line execution of [dGL99b] does not allow the automatic (off-line) verification of a plan. The idea is that instead of deriving a linear sequence of actions the interpreter should yield a *tree* of actions. A particular path in the tree then represents a legal execution of primitive actions conditioned on the possible outcome of sensing actions along the way. To represent such trees, Lakemeyer augments the situation calculus with *conditional action trees* (CAT). Besides primitive actions, a CAT may include branching nodes. A branching node has the form $[\phi, c_1, c_2]$, where ϕ is a formula which determines which branch is to be executed, and c_1 and c_2 are the true- and false-branch. Given a program that may contain sensing actions, the sGOLOG interpreter produces CAT's instead of sequences of actions in an off-line fashion. Here, the introduction of new branches in a CAT is left under the control of the user, who also has the responsibility to ensure that the branch conditions will be known at execution time.

Recently, Sardina [Sar01] has proposed to combine de Giacomo and Levesque's idea to interleave on-line and off-line execution with sGOLOG's notion of CAT's. In particular, he introduces a new operator Σ_c which allows for off-line search of conditional plans under user control.

DTGolog In [BRST00, Sou01], Boutilier, Reiter, Soutchanski and Thrun propose a decision-theoretic extension of GOLOG called DTGolog. DTGolog can be seen as a synthesis of MDPs and GOLOG. To represent the effects of noisy actions, DTGolog employs an action theory similar to the probabilistic situation calculus proposed by Pinto *et al.* [PSSM00]. In particular, stochastic agent actions are associated with a finite set of deterministic actions from which nature chooses stochastically. Additionally, a DTGolog domain specification includes an optimization theory which contains axioms specifying a reward function. A DTGolog program is written using the same instructions as in GOLOG, in particular nondeterministic instruc-

tions. Given a background theory, the semantics of a DTGolog program corresponds to that execution of the (nondeterministic) program which has the highest expected utility. Thus, from the point of view of MDPs, DTGolog programs can be seen as constraints on the space of allowable policies.

DTGolog assumes full observability of the domain. To "implement" full observability, a DTGolog background action theory includes a new class of *sense condition axioms*. These axioms associate a logical condition with each of nature's deterministic actions, which allows the agent to disambiguate the state using sensor information. The output of the DTGolog interpreter thus consists of the sequential composition of agent actions, sensing actions which serve to identify nature's choices, and conditionals. Intuitively, the agent senses the outcome of each stochastic action a (which deterministic action has been chosen by nature?), and then branches depending on the actual action which has occurred.

Knowledge-Based Programs In [Rei00], Reiter investigates the issue of knowledge based programming. Here, a knowledge-based program is a GOLOG program that includes sensing actions (in the sense of [SL93]) and that explicitly makes appeal to the agent's *knowledge* at execution time. Knowledge-based programs are intended to be executed on-line. Although they may involve nondeterministic instructions, they have to be designed in a way that prevents backtracking over sensing actions. The paper also shows how knowledge tests can be reduced to provability.

Further Work Related to GOLOG Poole [Poo96, Poo98] proposes an integration of decision theory, the independent choice logic [Poo97], and the situation calculus. Unlike the previous approaches, Poole's approach is not based on the standard predicate calculus semantics but on the concepts of *selector functions* [Poo97] and *stable models* [GL88]. In particular, an agent always knows what situation it is in, but the situation doesn't fully specify what is true in the world. To account for sensing, Poole introduces passive sensors which are represented by means of a set of terms called *observables* (there are no sensing actions in the sense of [SL93]). In any situation the robot has perfect knowledge of the value of the observables, and it can execute plans which are conditioned on the value of observables. The aim of this approach is to predict the probabilistic effects of a conditional plan, and in particular its expected utility.[2]

Thielscher has developed the action programming language FLUX [Thi00a], which provides similar facilities as GOLOG but is based on the fluent calculus. In [Thi01a], the language is extended to deal with knowledge and sensing, similarly to Reiter's knowledge based programs [Rei00]. Unlike Reiter's approach which is restricted to pure on-line execution, the extended FLUX interpreter is able to infer conditional plans including sensing actions. In [MT01], Martin and Thielscher present a planning and execution monitoring extension of FLUX based on a solution to the qualification problem in the fluent calculus [Thi01b], which allows recovering from unexpected action failures.

Based on Scherl and Levesque's representation of knowledge and sensing in the situation calculus [SL93], Levesque [Lev96] proposes the language $\mathcal{R}$ which includes loops and the

[2] As Poole states, "in [BHL95, BHL99] the probabilistic reasoning is internal to the agent", while in his framework "the agents [...] do not (have to) do probabilistic reasoning"; "an optimal agent may maintain a belief state [...] but it does not have to." Thus, Poole's approach is about probabilistic projection, not about belief update.

conditional construct $branch(a, r_1, r_2)$. The intuition is that $branch(a, r_1, r_2)$ first executes a binary sensing action a, and thereafter either executes the program r_1 or r_2 depending on the outcome of a. Finally, Shanahan [Sha98, SW00] has advocated a fully logic-based approach to robot control where a variety of tasks like planning, sensor integration, navigation and map building are all accomplished in an event-calculus framework.

2.3 Robot Controllers

In most existing robotics applications, the robot controllers are written using specialized, non-logic-based robot programming languages. Those languages provide features which are not present in the GOLOG family of languages and which have proven to be quite useful in practice. We will now take a look at the area of non-logic-based robot control and on the features provided by modern non-logic-based robot programming languages.

2.3.1 Robot Control Architectures

In the early days of AI-based robotics, the dominant control architecture for autonomous robots was the classical sense-plan-act (SPA) approach. This approach, which goes back to the work on the (first version) of the robot Shakey [Nil84], decomposed the control problem into three functional elements: a sensing system which translated sensor input into a world model, a planning system, and an execution system. This classical approach suffered from numerous shortcomings. In particular, planning and world modeling turned out to be very hard which caused the approach to have difficulties to react in real-time, and open-loop plan execution was inadequate in the face of the uncertainty encountered in realistic robot environments.

Several alternative approaches to robot control have been proposed to overcome the shortcomings of SPA. One the most prominent (and radical) departures from SPA is the behavior-based approach advocated by Rodney Brooks [Bro86, Bro91]. The behavior-based approach stresses the use of minimal internal representation, following the slogan "the world is its own model". Brooks proposed to decompose the problem of robot control into layers corresponding to different levels of behavior. Central to his approach is the concept of *subsumption*, which means that more complex layers can influence lower layers. Subsumption achieved dramatic success regarding basic-level tasks like obstacle avoidance or navigation. However, the approach had severe difficulties to scale to more complex tasks (cf. [Gat98]).

Today, most robot control architectures are hybrid systems that make use of both reactive components like behaviors, and components which are based on a symbolic representation. All of these approaches comprise two main layers: a reactive feedback control mechanism, and a task-control or *sequencer* layer which initiates and monitors behaviors, taking care of temporal aspects of coordinating behaviors. In addition, these architecture usually include a higher-level deliberative layer to provide look-ahead planning capabilities. The communication between the different layers is achieved via messages. Some of the most prominent hybrid control architectures are the 3T architecture [BFG+97], ATLANTIS [Gat92] and SSS [Con92]. But they also include the mobile robot Diablo [BFH+98], whose higher control level is based on the language $\mathcal{A}$, the very successful museum tour-guide robots RHINO [BBC+95, BCF+00] and MINERVA [TBB+99], as well as Saphira [KMRS97], XAVIER [SGH+97b] and NMR [PBC+97].

Low-Level Processes At the lowest layer, most hybrid architectures employ a suite of continuous processes (often called behaviors or skills) that provide situated control for the physical actions effected by the system. Here, we refer to such processes as *low-level processes*. We won't elaborate on the topic of low-level robot control except to note that this layer can be quite complex. It may involve a fuzzy controller [KMRS97] or may be based on POMDPs [SK95]. State-of-the-art basic-task levels like the BeeSoft system [BCF+00] used in the RHINO and MINERVA projects provide robust facilities for collision avoidance [FBT97], localization [FBT99, FBDT99], map building building and path planning [TBB+98].

2.3.2 Non-Logic-Based Robot Programming Languages

In most of these architectures, it is the sequencer that plays the role of the main executive, and which (eventually) invokes the planner to accomplish goals. The sequencer must support continuous processes and concurrent actions, since a robot must generally perform many activities. While parts of the sequencer programs may automatically be generated by a planner, quite often they are hand-tailored by a robot programmer. As stated by Kurt Konolige [Kon97], "when one thinks of writing robot programs, it is sequencer programs that are the result." We will now briefly go over some of the most prominent robot programming languages and then elaborate on McDermott's reactive plan language RPL [McD91].

PRS-Lite PRS-Lite [Mye96] is a derivative of the family of Procedural Reasoning Systems [GI89] optimized for compactness and efficiency. PRS-Lite provides constructs like test instructions which provide the ability to test environment variables, execute instructions used to set environment variables or to trigger external actions, intend and unintend instructions used to activate respectively to explicitly terminate low-level processes, and wait-for instructions that enable limited suspension until a certain condition or event occurs. The wait-for modality is critical in that it enables synchronization among concurrent processes. Sequencing constructs like if (conditional activation) and and (parallel activation) allow goal-ordering. A compiler transforms the specifications into finite-state machines, which allows the realization of an execution loop with a very short cycle time.

Colbert Colbert [Kon97] is a successor of the PRS-Lite system which provides similar instructions. A major difference between PRS-Lite and Colbert is that the syntax of the latter is based on ANSI C, which facilitates the readability of robot control programs for programmers used to sequential, conditional and iterative constructs. Nevertheless, it provides much the same functionality as PRS-Lite, including iteration, concurrent activities and suspension via waitfor. Just like PRS-Lite, Colbert does not include capabilities for deliberation over different procedures.

RAP The RAP system [Fir87, Fir95] is a plan and task representation based on program-like reactive action packages, or RAPs. The idea is that a planner produces "sketchy" plans, relatively vague plans where some steps must be determined at run-time. The RAP execution system then fills in the details of the sketchy plan at run time. The RAPs thus serve as a bridge between the planner and the control system. They expand vague plan steps into detailed instructions by choosing an appropriate method for the next step from a preexisting library. To determine what method to use, it maintains a memory containing what is believed to be true about the world. If a method does not achieve its intended goal, the RAP system

chooses another method and tries again. Like PRS-Lite and Colbert, RAP provides facilities for concurrent execution and synchronization (in particular, the wait-for construct). The RAP system can thus be seen in two different ways: as a reactive system which can accomplish quite complex goals given the methods in the library, or, on the other hand, as supplying a planner with abstract "primitive actions."

RPL RPL [McD91, McD92a, Bee99] is a notation for writing reactive plans for agents, e.g. for robots. Recently, it has been used successfully to control the behavior of the mobile robot MINERVA deployed in a realistic environment for an extended period of time [TBB+99]. RPL's immediate ancestor is the RAP notation. Compared to RAP, however, it does not attempt to maintain a world model that tracks the situation outside the robot. The intent of RPL is to spell out how the behavior of an agent is to be driven by events around it, and to do so in a transparent notation.

RPL is implemented in LISP, and RPL programs look like LISP programs. RPL provides the constructs (seq c_1 ... c_n), (if cond c_{then} c_{false}) and (loop cond -body-), which allow, respectively, sequential execution of several sub-plans, conditional execution depending on the truth-value of cond, and while-loop iteration. Local variables can be declared by the usual LISP construct (let ((v_1 val_1) (v_n val_n)) -body-), which introduces new variables v_i with initial value val_i. Finally, RPL code affects the outside world by special RPL procedures that send signals to the output ports.

As mentioned above, the intent of RPL is to specify how the behavior of an agent is to be driven by events around it. A key mechanism to achieve this aim is the concept of a *fluent*. Here, the term *fluent* is used to refer to a program variable of time-varying value which is set by sensors. The RPL run-time system takes care of automatically updating the value of fluents when new sensor inputs arrive. For example, a reactive controller written in RPL will make use of fluents like robot-x and robot-y to represent the robot's x and y coordinates. We stress that in RPL fluents are merely a special kind of program variable, and should not be confused with the situation calculus notion of a fluent.

An RPL plan can react to changes of the values of fluents by means of the RPL instruction (wait-for fl), where fl is a binary fluent. This construct causes the execution to suspend if the truth value of fl is false, and to wait until it becomes true. The wait-for instruction can not only be applied to primitive fluents, but also to fluent-networks, boolean expressions involving fluents. Figure 2.1 illustrates an example fluent network which verifies that the robot's position lies in a certain range. Using wait-for and such a fluent-network, it is possible to specify a robot program that reacts as soon as the robot enters a specific area.

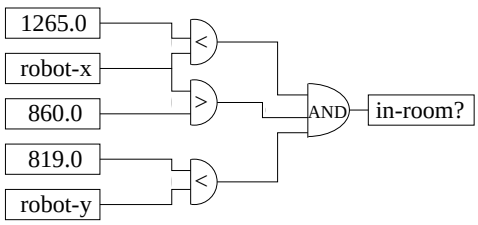


Figure 2.1: A fluent network

RPL supports true concurrency, meaning that several sub-programs can be executed in parallel. In particular, RPL provides the constructs (par $a_1...a_n$) and (try-all $a_1...a_n$), which both start executing the sub-plans $a_1...a_n$ in parallel. The difference between the two construct is that the former ends only after all sub-plans $a_1...a_n$ have ended, while the latter is completed as soon as the first sub-plans ends.

Programs running concurrently often compete for resources. For example, different processes may compete for the control of the robot's wheels. In order to synchronize concurrent

programs that should not execute simultaneously, RPL provides a type of semaphore called a *valve*. Whenever two procedures should not run concurrently, they are arranged such that they compete for a valve. The loser waits for the winner to finish and release it.

It is also possible to execute sub-plans with different priorities. A process with a higher priority can preempt a valve from a process with a lower priority. The construct (with-policy pol body) indicates that the two procedures pol and body are to be executed in parallel, and that pol has a higher priority than body. This construct is useful if a secondary plan, the policy, is executed solely to constrain the execution of the plan body. For example, a robot might go from one place to another, but adopt the policy that if its gripper should ever become empty, then it must stop and pick up the object it dropped. Note that here policies are just ordinary plans, which should not be confused with the notion of a policy in (PO)MDPs.

The XFRM System The XFRM system of Beetz and McDermott [McD92b, BM94, BM97] is a planning framework which provides powerful and general tools for execution monitoring and prediction-based revision of RPL plans. In particular, it includes the projection module PTOPA [McD94] that generates symbolic representations of possible execution scenarios. PTOPA takes as its input an RPL plan, rules for generating exogenous events, and a set of probabilistic rules describing the effects of exogenous events and low-level processes. PTOPA randomly samples execution scenarios from the probability distribution that is implied by the rules. In [BG98], the framework is extended to allow the projection of RPL plans interacting with low-level processes involving continuous change.

Unlike the GOLOG family of languages discussed in the next section, the XFRM projection mechanism does not rely on a perspicuous declarative semantics. Although Davis [Dav92] has presented a formal semantics for a subset of the language RPL that relates a task's starting time, its completion, and the primitive actions that it executes to those of its subtasks, this formalism has not been connected to the PTOPA projection module. On the other hand, [McD94] provides a formal semantics for PTOPA, which however is restricted to plans that consist of a totally ordered sequence of primitive actions. Finally, [BG00] partly formalizes the extension of XFRM for dealing with continuous change.

Other Approaches to Robot Control Other approaches to robot control include the following: The language L [Bro93] is a dialect of LISP, which is tailored for robot control. In particular, it can run a LISP system in an embedded systems with 10 KB of memory, complete with garbage collector, and provides multi threading facilities to support concurrent execution. [AI99] describes a logic-based control architecture based on subsumption. The approach includes a set of first-order logic theories for each layer, and makes use of non-monotonic reasoning (circumscription) to model the connections between the layers. Finally, [FLK93] and [LK92] propose robot control architectures based on a Blackboard control unit.

2.4 Discussion

As described in Section 2.1, there exists a large body of sophisticated formalisms for reasoning about action and change, including the situation calculus. While (some of) the other approaches to reasoning about action and change are similarly expressive or sometimes even superior to the situation calculus, the situation calculus stands out in that there exists an advanced body of work on the integration of expressive high-level programming languages into it,

namely the GOLOG family described in Section 2.2. GOLOG and its derivatives allow the specification of high-level robot behavior on a high level of abstraction, provide powerful projection capabilities based on a perspicuous formalism for reasoning about action and change, and have already been used to control real robots in first successful experiments [LTJ98, BCF+00].

However, in its current form the GOLOG family of logic-based languages lacks the expressiveness provided by non-logic-based robot programming languages like RPL, RAP or Colbert discussed in Section 2.3.1. In particular, the GOLOG family does not provide facilities for dealing with continuous change, event-driven behavior, low-level processes or probabilistic uncertainty. On the other hand, the non-logic-based languages do not provide the possibility to project plans based on a perspicuous, declarative semantics. In the remainder of this thesis, we will investigate how the GOLOG framework can be extended so as to shorten the gap in expressiveness between non-logic-based and logic-based robot control languages, thus making GOLOG more suitable for realistic applications.

Chapter 3

The Situation Calculus and ConGolog

In this chapter, we will introduce the logical framework on which the work presented in this thesis is based. In particular, we formally describe the situation calculus [McC63, LPR98], a logical language for reasoning about dynamically changing worlds, and ConGolog [dGLL97, dGLL00], a high-level programming language based on the situation calculus. Here, we use a version of the situation calculus developed at the University of Toronto (cf. [Rei01]). Finally, we describe how the situation calculus can be extended to represent probabilistic uncertainty about the state of the world, following [BHL95, BHL99].

A word on the notation: we use the standard logical connectives "$\neg$" (negation), "$\wedge$" (conjunction), "$\vee$" (disjunction), "$\supset$" (implication), "$\equiv$" (equivalence), "$\forall$" (universal quantification), and "$\exists$" (existential quantification). In logical languages, when confusions may arise a quantifier's scope must be indicated explicitly with parentheses. As an alternative, we often use the "dot" notation which indicates that the quantifier preceding the dot has maximum scope. So $\forall x.P(x) \supset Q(x)$ stands for $(\forall x)[P(x) \supset Q(x)]$. Likewise, we often omit parenthesis by assuming that $\wedge$ takes precedence over $\vee$, so that $P \wedge Q \vee R \wedge S$ stands for $(P \wedge Q) \vee (R \wedge S)$. $\supset$ and $\equiv$ bind with lowest precedence. In formulas involving second-order quantification over predicate or function variables, we leave the arity of the second-order variables implicit. For example, we simply write $\forall P.(\forall x)[P(x)] \supset (\exists y)[P(y)]$. The arity will become clear from the context. Instead of an expression $\exists x_1.\exists x_2.\ ...\exists x_n.P(x_1, ...x_n)$ we often use the abbreviation $\exists x_1, ...x_n.P(x_1, ...x_n)$, and analogously for universal quantification. Similarly, we sometimes abbreviate sequences $x_1, ..., x_n$ of pairwise different variables as $\vec{x}$, and use the notation $\vec{x} = \vec{y}$ as a shorthand for $x_1 = y_1 \wedge ... \wedge x_n = y_n$. Finally, we use the convention that all free variables are implicitly universally quantified, so $\exists y.y > x$ stands for $\forall x.\exists y.y > x$.

3.1 The Situation Calculus

The situation calculus [McC63, LPR98] is a sorted second-order language with equality, designed for representing and reasoning about dynamically changing worlds. The intuition behind the situation calculus is that initially, the world is in an *initial situation*, and that the world evolves from one situation to another as the result of the execution of *primitive actions*. There are three disjoint sorts: *actions*, *situations*, and ordinary *objects*. There are

two functions of sort *situation*: a special constant S_0 used to denote the initial situation, namely that situation in which no actions have yet occurred, and a binary function symbol *do* where $do(a, s)$ denotes the successor situation of s resulting from performing action a in s. For convenience, we abbreviated nested *do* expression of the form $do(a_n, \cdots do(a_1, S_0) \cdots)$ as $do([a_1, \ldots, a_n], S_0)$.

Actions may be parameterized. For example, $gotoRoom(r)$ might stand for the action of a robot traveling to room r. Accordingly, $do(gotoRoom(R_1), s)$ denotes the situation resulting from the robot traveling to Room R_1 when the world is in situation s. The state of the world in a particular situation s is characterized by relational and functional *fluents*, relations and functions taking a situation term as their last argument. For example, $CoffeeRequest(r, s)$ might mean that in situation s the robot is requested to deliver coffee to room r. Similarly, the functional fluent $robotLoc(s)$ might represent the room in which the robot is located in situation s.

The applicability and the effects of actions are characterized by certain types of axioms. First, there is a binary predicate symbol $Poss : action \times situation$, which intuitively represents whether an action is physically possible in a certain situation. In order to state the necessary and sufficient conditions for an action a, we adopt the idealizing approach to write an *action precondition axiom* of the form $Poss(a(\vec{x}), s) \equiv \Phi(\vec{x}, s)$, where $\vec{x}$ stands for the arguments of a.[1] As an example, the action precondition axiom $Poss(giveCoffee(r), s) \equiv robotLoc(s) = r$ says that in any situation s the robot can serve coffee in room r if and only if the robot is in room r in situation s.

Besides specifying *action precondition axiom*, one must specify how the actions affect the state of the world. This can be done by writing positive and negative *effect axioms*. Positive respectively negative effect axioms have the following form:

$$Poss(A, s) \wedge \phi^+(\vec{x}, s) \supset F(\vec{x}, do(A, s));$$
$$Poss(A, s) \wedge \phi^-(\vec{x}, s) \supset \neg F(\vec{x}, do(A, s)).$$

Here, ϕ^+ and ϕ^- are first-order formulas specifying the conditions under which the action A will have the effect of causing the fluent F to become true respectively false. For example, the positive effect axiom

$$Poss(giveCoffee(r), s) \wedge CoffeeRequest(r, s) \supset happy(r, do(giveCoffee(r), s))$$

says that serving coffee in a room r in which coffee is requested causes the people in room r to be happy. Similarly, the negative effect axiom

$$Poss(giveCoffee(r), s) \supset \neg CoffeeRequest(r, do(giveCoffee(r), s))$$

says that after serving coffee in a room r the robot is no longer requested to serve coffee in r (in this axiom the condition ϕ^- corresponds to TRUE). Effect axioms thus provide the "causal laws" for the domain.

Unfortunately, in general these types of axioms are not sufficient to reason about change. The problem is that one also needs to say which fluents are *not affected* by the performance of an action. For example, in order to deduce that serving coffee does not affect the robot's position one needs axioms like $robotLoc(s) = r \supset robotLoc(do(giveCoffee(r), s)) = r$. Such

[1] Note that by this idealizing approach we simply ignore the qualification problem (cf. Section 2.1.1).

axioms, which state the "non-effects" of actions are called *frame axioms*. The so-called *frame problem* arises because the number of these frame axioms is very large, in general of the order of $2 \times \mathcal{A} \times \mathcal{F}$, where $\mathcal{A}$ is the number of actions and $\mathcal{F}$ the number of fluents. This complicates the task of axiomatizing a domain and can make theorem proving extremely inefficient. In a nutshell, the frame problem corresponds to the question how to represent the effects of actions without having to explicitly represent all their non-effects. It was first brought to light by McCarthy and Hayes in the late Sixties [MH69].

3.1.1 A Simple Solution to the Frame Problem (Sometimes)

To deal with the frame problem, we use a monotonic approach due to Ray Reiter [Rei91], which is based on earlier proposals by Pednault, Haas, and Schubert [Ped89, Haa87, Sch90]. The solution only applies to deterministic actions, hence the caveat "sometimes" in the title.[2] The approach is based on the idea to collect all effect axioms about a given fluent and make a completeness assumption, that is assume that they specify all of the ways that the value of the fluent may change. A syntactic transformation is then applied to obtain a *successor state axiom* for the fluent. For example, the following is a successor state axiom for the fluent $CoffeeRequest(r, s)$:

$$Poss(a, s) \supset [CoffeeRequest(r, do(a, s)) \equiv a = newRequest(r) \vee CoffeeRequest(r, s) \wedge a \neq giveCoffee(r)].$$

We will now describe how Reiter's solution to the frame problem works for relational fluents; functional fluent can be handled similarly. The first step consists of rewriting all positive effect axioms for a given fluent as a single, logically equivalent positive effect axiom with the following syntactic normal form:

$$Poss(a, s) \wedge \gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)). \tag{3.1}$$

Similarly, one rewrites the negative effect axioms into the following normal form:

$$Poss(a, s) \wedge \gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)). \tag{3.2}$$

This transformation into normal form can be done automatically as follows: each of the given positive effect axioms has the form:

$$Poss(A, s) \wedge \phi_F^+ \supset F(\vec{t}, do(A, s))$$

where A is an action term (like $giveCoffee(r)$) and the $\vec{t}$ are terms. One then rewrites the positive effect axiom in the following, logically equivalent form:

$$Poss(a, s) \wedge a = A \wedge \vec{x} = \vec{t} \wedge \phi_F^+ \supset F(\vec{x}, do(a, s)).$$

Here, $\vec{x}$ are new variables distinct from one another and from any variable occurring in the original effect axiom. Now suppose $y_1, ..., y_m$ are all the free variables occurring in the original effect axioms except for the situation variable s. These variables are thus implicitly universally quantified in the above axiom, and one can rewrite it in the following, logically equivalent form:

[2] In particular, the solution does not apply to nondeterministic action with disjunctive effects, like for example a coin toss that can result in either heads or tails: $Poss(toss, s) \supset heads(do(toss, s)) \vee tails(do(toss, s))$.

$$Poss(a, s) \wedge [\exists y_1, ..., \exists y_m . a = A \wedge \vec{x} = \vec{t} \wedge \phi_F^+] \supset F(\vec{x}, do(a, s)).$$

So each positive effect axiom for fluent F can be rewritten in the logically equivalent form:

$$Poss(a, s) \wedge \Phi_F \supset F(\vec{x}, do(a, s))$$

where Φ_F is a formula whose free variables are among $\vec{x}, a, s$. If one does this for each of the k positive effect axiom for F, one gets

$$Poss(a, s) \wedge \Phi_F^{(1)} \supset F(\vec{x}, do(a, s))$$
$$\overset{...}{Poss(a, s) \wedge \Phi_F^{(k)}} \supset F(\vec{x}, do(a, s)).$$

These sentences can be rewritten as a single, logically equivalent sentence

$$Poss(a, s) \wedge [\Phi_F^{(1)} \vee ... \vee \Phi_F^{(k)}] \supset F(\vec{x}, do(a, s))$$

which is the normal form for the positive effect axiom for fluent F. The normal form for negative effect axioms can be computed similarly.

Next, one makes the following *causal completeness assumption*: *The two axioms 3.1 and 3.2 characterize all the conditions under which action a can lead to F becoming true (respectively false) in the successor situation.* Hence, if F's truth value changes from FALSE in situation s to TRUE in the next situation $do(a, s)$, then $\gamma_F^+(\vec{x}, a, s)$ must have been true. Similarly, if F's truth value changes from TRUE to FALSE then $\gamma_F^-(\vec{x}, a, s)$ must have been true. The *causal completeness assumption* can thus be formally specified by the following *explanation closure axioms*:

$$Poss(a, s) \wedge F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a, s)) \supset \gamma_F^-(\vec{x}, a, s), \tag{3.3}$$
$$Poss(a, s) \wedge \neg F(\vec{x}, s) \wedge F(\vec{x}, do(a, s)) \supset \gamma_F^+(\vec{x}, a, s). \tag{3.4}$$

To see how explanation closure axioms function like frame axioms, we rewrite them in the logically equivalent form:

$$Poss(a, s) \wedge F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)),$$
$$Poss(a, s) \wedge \neg F(\vec{x}, s) \wedge \neg\gamma_F^+(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)).$$

To make this work, one needs *unique names axioms for actions*: for distinct action names A and B,

$$A(\vec{x}) \neq B(\vec{x}),$$
$$A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}.$$

Reiter shows that if T is a first-order theory that entails $\neg\exists \vec{x}, a, s . Poss(a, s) \wedge \gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s)$ then T entails that the effect axioms in normal form 3.1 and 3.2 together with the explanation closure axioms 3.3 and 3.4 are logically equivalent to:

$$Poss(a, s) \supset [F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)].$$

The above formula is called the *successor state axiom for the relational fluent* F. Similarly, a *successor state axiom for a functional fluent* f has the following form [Rei01]:

$$Poss(a, s) \supset [f(\vec{x}, do(a, s)) = y \equiv \gamma_f(\vec{x}, y, a, s) \vee f(\vec{x}, s) = y \wedge \neg\exists y'.\gamma_f(\vec{x}, y', a, s)]. \quad (3.5)$$

Here, $\gamma_f(\vec{x}, y, a, s)$ is a first order formula whose free variables are among $\vec{x}, y, a, s$. Intuitively, $\gamma_f(\vec{x}, y, a, s)$ specifies the conditions under which action a has the effect of causing f to get value y. In order to guarantee the consistency of a successor state axiom for a functional fluent, the following sentence must be entailed by the background theory:

$$\neg\exists \vec{x}, y, y', a, s.\gamma_f(\vec{x}, y, a, s) \wedge \gamma_f(\vec{x}, y', a, s) \wedge y \neq y'. \quad (3.6)$$

Note that this approach results in exactly $\mathcal{F}$ successor state axioms (together with $\mathcal{A}$ action precondition axioms plus the unique names axioms), compared to the $2 \times \mathcal{A} \times \mathcal{F}$ explicit frame axioms that would otherwise be required. Summarizing, this solution to the frame problem relies on the quantification over actions and on the causal completeness assumption. We remark that this approach is monotonic once the additions to the original axiom set have been performed, but as seen from the original axioms it is a non-monotonic method.

3.1.2 Basic Action Theories

Based on Reiter's solution of the frame problem, one can formulate a *basic action theory* [LPR98] which describe how the world changes as the result of the available actions. A basic action theory has the following form:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each relational fluent F, stating under what conditions the relational fluent $F(\vec{x}, do(a, s))$ holds as a function of what holds in situation s. Similarly, successor state axioms for each functional fluent f, stating under what conditions $f(\vec{x}, do(a, s)) = y$ holds.
- Unique names axioms for the primitive actions.
- Foundational, domain independent axioms:
  1. $S_0 \neq do(a, s)$;
  2. $do(a, s) = do(a', s') \supset a = a' \wedge s = s'$;
  3. $\forall P.P(S_0) \wedge [\forall s \forall a.(P(s) \supset P(do(a, s)))] \supset \forall s P(s)$;
  4. $\neg(s \sqsubset S_0)$;
  5. $s \sqsubset do(a, s') \equiv s \sqsubseteq s'$, where $s \sqsubseteq s'$ stands for $(s \sqsubset s') \vee (s = s')$;
  6. $s \prec s' \equiv s \sqsubset s' \wedge \forall a, s^*.s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*)$.

The first three foundational axioms serve to characterize the space of all situations, making it isomorphic to the set of ground terms of the form $do(a_n, \cdots do(a_1, S_0) \cdots)$. The third axiom ensures, by second-order induction, that there are no situations other than those accessible using $do(a, s)$ from S_0. As described in [Rei93, PR99], this induction axiom is important to

prove sentences that are universally quantified over states. The two axioms 4 and 5 serve to characterize the binary predicate $\sqsubset$: *situation* $\times$ *situation*, which defines an ordering relation on situations. Intuitively, $s \sqsubset s'$ holds if s' can be obtained from s by performing a finite number of actions. Finally, the last axiom defines the relation $\prec$, where $s \prec s'$ means that s is an initial subsequence of s' and that all the actions of s' following those of s can be executed one after the other. We call a situation s such that $(S_0 \prec s)$ *executable*. A situation s such that $\neg(S_0 \prec s)$ is a "ghost state," meaning that is is not reachable from S_0 by any executable sequence of actions. As with $\sqsubseteq$, we will use $s \preceq s'$ is a shorthand for $s \prec s' \vee s = s'$.

3.1.3 An Example

To illustrate the use of the situation calculus, let us consider a simple coffee delivery application, where a robot is to deliver coffee in an office environment with four rooms R_1, R_2, R_3, and R_4. In this domain, the state of the world can be characterized by the functional fluent *robotLoc*(s), which represents the room the robot is currently in, and the relational fluent *CoffeeRequest*(r, s), stating whether or not the robot is to deliver coffee to room r. The robot can travel to room r by means of the primitive action *gotoRoom*(r), and satisfy a coffee request by executing *giveCoffee*(r) in room r.

The following axioms, together with the foundational axioms and unique names axioms for primitive actions, represent a basic action theory modeling our coffee delivery application:

$$robotLoc(S_0) = R_1 \wedge CoffeeRequest(r, S_0) \equiv [r = R_2 \vee r = R_4]; \tag{3.7}$$

$$Poss(gotoRoom(r), s) \equiv \text{TRUE}; \tag{3.8}$$

$$Poss(giveCoffee(r), s) \equiv robotLoc(s) = r; \tag{3.9}$$

$$\begin{aligned} robotLoc(do(a, s)) = r \equiv\ & a = gotoRoom(r) \vee \\ & a \neq gotoRoom(r) \wedge robotLoc(s) = r; \end{aligned} \tag{3.10}$$

$$CoffeeRequest(r, do(a, s)) \equiv CoffeeRequest(r, s) \wedge a \neq giveCoffee(r). \tag{3.11}$$

In this example, $do([gotoRoom(R_2), giveCoffee(R_2)], S_0)$ denotes the situation resulting from first traveling to room R_2 and then serving coffee. It is straightforward to show that this situation is executable, i.e. that $S_0 \prec do([gotoRoom(R_2), giveCoffee(R_2)], S_0)$ holds.

Proof: (Sketch) The first action, $gotoRoom(R_2)$, is always possible (Axiom 3.8). From the successor state axiom for *robotLoc* (Axiom 3.10), we can conclude that after execution of that action the robot is in room R_2, i.e. $robotLoc(do(gotoRoom(R_2), S_0)) = R_2$. From this fact, and the action precondition for *giveCoffee* (Axiom 3.9), one can conclude that the second action, $giveCoffee(R_2)$, is possible in $do(gotoRoom(R_2), S_0)$. □

Similarly, it is straightforward to show that after traveling to room R_2 and serving coffee the coffee request for Room R_2 has been fulfilled. Formally, the above statement can be expressed as $\neg CoffeeRequest(R_2, do([gotoRoom(R_2), giveCoffee(R_2)], S_0))$, which follows directly from *CoffeeRequest*'s successor state axiom (Axiom 3.11).

3.2 ConGolog

ConGolog [dGLL97, dGLL00], a derivative of GOLOG [LRL+97], is a special action programming language based on the situation calculus. In particular, every primitive action in a

ConGolog program is an action of the underlying situation calculus domain theory. As a result, it is possible to project the outcome of a program, that is, to reason about how the world evolves when a program is executed. Thus, we use the terms complex action, program and plan interchangeably, following McDermott [McD92a] who takes plans to be programs whose execution can be reasoned about by the agent who executes the program.

Besides primitive actions, there is another class of primitive instructions: tests of the form $\phi?$, where ϕ stands for a situation calculus formula. The intuition behind $\phi?$ is to block if ϕ is false in the actual situation, and else continue with execution. Using tests, a program can query ConGolog's situation calculus model of the environment. The primitive instructions can be composed by constructs such as sequences, iterations and recursive procedures to define complex actions. ConGolog also provides nondeterministic instructions which allow the specification of nondeterministic programs. In addition to the sequential constructs already present in GOLOG [LRL+97], concurrent actions are introduced with a conventional interleaving semantics. Altogether, ConGolog provides the following deterministic and nondeterministic constructs:[3]

nil	empty program
α	primitive action
$\phi?$	wait/test action
$[\sigma_1, \sigma_2]$	sequence of two programs σ_1 and σ_2
$\mathsf{if}(\phi, \sigma_1, \sigma_2)$	conditional execution of σ_1 or σ_2
$\mathsf{while}(\phi, \sigma)$	loop
$\sigma_1 \rangle\rangle \sigma_2$	prioritized concurrent execution
$\sigma_1 \vert \sigma_2$	nondeterministic choice between actions
$\pi x.\sigma$	nondeterministic choice of arguments
σ^*	nondeterministic iteration
$\sigma_1 \mid\mid \sigma_2$	concurrent execution
$\sigma^{\mid\mid}$	concurrent iteration
$\{\mathsf{proc}(P_1(\vec{v_1}), \beta_1); ...; \mathsf{proc}(P_n(\vec{v_n}), \beta_n); \sigma\}$	procedures.

Here, nil is the empty program, which denotes the fact that nothing remains to be done. ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation. For example, $\mathit{CoffeeRequest}(R_1, \mathit{now})?$ will block if ϕ is false in the *actual* situation in which the test is to be executed, and else succeed immediately. Similarly, α stands for a situation calculus action where the special situation constant *now* may be used to refer to the current situation. Using *now* in primitive actions is convenient to parameterize the action with the value of a functional fluent. For example, $\mathit{giveCoffee}(\mathit{robotLoc}(\mathit{now}))$ means that the robot is to serve coffee in the room it is actually in. When no confusions arise, we will simply leave out the *now* argument from the fluents altogether, e.g. write $\mathit{CoffeeRequest}(R_1)$ as an abbreviation for $\mathit{CoffeeRequest}(R_1, \mathit{now})$. Besides, we will use $[\alpha, \beta, \gamma]$ as a shorthand for $[\alpha, [\beta, \gamma]]$, and similarly write $\mathsf{if}(\phi, \alpha)$ instead of $\mathsf{if}(\phi, \alpha, \mathsf{nil})$.

The semantics of the deterministic constructs $[\sigma_1, \sigma_2]$, $\mathsf{if}(\phi, \sigma_1, \sigma_2)$, $\mathsf{while}(\phi, \sigma)$ and proc correspond, roughly, to the their usual semantics in ordinary programming languages. However, we remark that if and while are synchronized in the sense that testing the condition ϕ

[3] [dGLL00] additionally considers the interrupt construct $< \phi \rightarrow \sigma >$, which however is only a macro defined in terms of the other constructs.

and executing the first action of the branch chosen are executed as an atomic unit.[4] $\sigma_1 \rangle\rangle \sigma_2$ denotes the concurrent execution of the actions of σ_1 and σ_2, but with σ_1 having a higher priority than σ_2. This means that σ_2 may only execute when σ_1 is either done or blocked; whenever σ_1 can execute, it *blocks* the execution of σ_2.

Besides these deterministic instructions, ConGolog provides the nondeterministic constructs $(\sigma_1 \mid \sigma_2)$, $(\pi x.\sigma)$, (σ^*), $(\sigma_1 \mid\mid \sigma_2)$ and $(\sigma^{||})$. The idea of the first three constructs is, respectively, to nondeterministically execute either σ_1 or σ_2; to nondeterministically pick an individual x, and for that x perform the program σ; and to execute σ an arbitrary number of times. The $\sigma_1 \mid\mid \sigma_2$ construct denotes un-prioritized concurrent execution. $\sigma_1 \mid\mid \sigma_2$ is a nondeterministic instruction in that it allows an arbitrary interleaving of the actions of σ_1 and σ_2. Finally, the $\sigma^{||}$ construct is like nondeterministic iteration, but where the instances of σ are executed concurrently, rather than in sequence.

3.2.1 A Transition Semantics

Formally, the semantics of ConGolog is defined using a so-called transition semantics, which defines single steps of computation. There is a relation, denoted by the predicate $Trans(\sigma, s, \delta, s')$, that associates with a given program σ and situation s a new situation s' that results from executing σ's first action in s, and a new program δ that represents what remains of σ after having performed that action. Furthermore, one needs to define which configurations $\langle \sigma, s \rangle$ are final, meaning that the computation can be considered completed when a final configuration is reached. This is denoted by the predicate $Final(\sigma, s)$. Note that the use of a transition semantics necessitates the reification of programs as first order terms in the logical language (cf. [dGLL00] and Appendix A). To simplify the discussion, we postpone the introduction of procedures which necessitates the use of second-order logic.

The predicate *Final* is characterized by the following set of axioms. The term $\phi[s]$ denotes the formula obtained by substituting the situation variable s for all occurrences of *now* in fluents appearing in ϕ (cf. [dGLL00] and Appendix A).

$Final(\alpha, s) \equiv$ FALSE , where α is a primitive action

$Final(nil, s) \equiv$ TRUE, where *nil* is the empty program

$Final(\phi?, s) \equiv$ FALSE

$Final([\sigma_1, \sigma_2], s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$

$Final(\mathsf{if}(\phi, \sigma_1, \sigma_2), s) \equiv \phi[s] \wedge Final(\sigma_1, s) \vee \neg\phi[s] \wedge Final(\sigma_2, s)$

$Final(\mathsf{while}(\phi, \sigma), s) \equiv \neg\phi[s] \vee Final(\sigma, s)$

$Final(\sigma_1 \rangle\rangle \sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$

$Final(\sigma_1 | \sigma_2, s) \equiv Final(\sigma_1) \vee Final(\sigma_2)$

$Final(\pi x.\sigma, s) \equiv \exists v.Final(\sigma_v^x, s)$

$Final(\sigma^*, s) \equiv$ TRUE

$Final(\sigma_1 \mid\mid \sigma_2) \equiv Final(\sigma_1) \wedge Final(\sigma_2)$

$Final(\sigma^{||}, s) \equiv$ TRUE

[4] We remark that non-concurrent GOLOG [LRL+97] considers non-synchronized versions of the if-then-else and loop constructs.

Let us first consider when a deterministic program is final. A program that consists of a primitive action α or a test $\phi?$ is never final. On the other hand, the empty program nil is always final. A conditional $\mathsf{if}(\phi, \sigma_1, \sigma_2)$ is final if ϕ holds and σ_1 is final, or if ϕ is false and σ_2 is final. A sequence $[\sigma_1, \sigma_2]$ is final if and only if both σ_1 and σ_2 are final. A $\mathsf{while}(\phi, \sigma)$ loop is final if ϕ is false or if σ is final. The concurrent, prioritized execution of two programs is final if and only if both programs are final. As for the nondeterministic instructions, $\sigma_1|\sigma_2$ is final if any nondeterministic branch σ_i is final. $\pi x.\sigma$ is final if there exists a binding v for x such that σ_v^x is final, where σ_v^x is obtained from σ by substituting x with v. σ^* is final, since it is allowed to execute 0 times. Similarly for $\sigma^{||}$. Finally, $\sigma_1 \mid\mid \sigma_2$ is final if and only if both σ_i are final.

The following set of axioms characterizes the predicate *Trans*. Similar to $\phi[s]$, $\alpha[s]$ denotes the action obtained by substituting the situation variable s for all occurrences of *now* in functional fluents appearing in α (cf. [dGLL00] and Appendix A).

$$Trans(nil, s, \delta, s') \equiv \textsc{False}$$

$$Trans(\alpha, s, \delta, s') \equiv Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s)$$

$$Trans(\phi?, s, \delta, s') \equiv \phi[s] \wedge \delta = nil \wedge s' = s$$

$$Trans([\sigma_1, \sigma_2], s, \delta, s') \equiv$$
$$\exists\gamma. Trans(\sigma_1, s, \gamma, s') \wedge \delta = [\gamma, \sigma_2] \vee Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s')$$

$$Trans(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') \equiv$$
$$\phi[s] \wedge Trans(\sigma_1, s, \delta, s') \vee \neg\phi[s] \wedge Trans(\sigma_2, s, \delta, s')$$

$$Trans(\mathsf{while}(\phi, \sigma), s, \delta, s') \equiv$$
$$\exists\gamma. Trans(\sigma, s, \gamma, s') \wedge \phi[s] \wedge \delta = [\gamma, \mathsf{while}(\phi, \sigma)]$$

$$Trans(\sigma_1 \rangle\rangle \sigma_2, s, \delta, s') \equiv$$
$$\exists\gamma.\delta = (\gamma \rangle\rangle \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee$$
$$\exists\gamma.\delta = (\sigma_1 \rangle\rangle \gamma) \wedge Trans(\sigma_2, s, \gamma, s') \wedge \forall\gamma', s''.\neg Trans(\sigma_1, s, \gamma', s'')$$

$$Trans(\sigma_1|\sigma_2, s, \delta, s') \equiv Trans(\sigma_1, s, \delta, s') \vee Trans(\sigma_2, s, \delta, s')$$

$$Trans(\pi x.\sigma, s, \delta, s') \equiv \exists v. Trans(\sigma_v^x, s, \delta, s)$$

$$Trans(\sigma^*, s, \delta, s') \equiv \exists\gamma.\delta = [\gamma, \sigma^*] \wedge Trans(\sigma, s, \gamma, s')$$

$$Trans(\sigma_1 \mid\mid \sigma_2, s, \delta, s') \equiv$$
$$\exists\gamma.\delta = (\gamma \mid\mid \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee \exists\gamma.\delta = (\sigma_1 \mid\mid \gamma) \wedge Trans(\sigma_2, s, \gamma, s')$$

$$Trans(\sigma^{||}, s, \delta, s') \equiv \exists\gamma.\delta = (\gamma \mid\mid \sigma^{||}) \wedge Trans(\sigma, s, \gamma, s')$$

As before, we first consider the deterministic instructions. Intuitively, a program that consists of a single atomic action α in a situation s results in the execution of $\alpha[s]$ with an empty remaining program if and only if $\alpha[s]$ is executable in s. A test $\phi?$ succeeds if $\phi[s]$ holds, leaving nothing to be done, or is blocked, meaning that it cannot result in a transition. The execution of $[\sigma_1, \sigma_2]$ in s may result in any successor situation that could be reached by the execution of σ_1, with remaining program $[\gamma, \sigma_2]$, where γ is what remains of σ_1; or, if σ_1 is final in s, it just corresponds to the execution of σ_2. The execution of $\mathsf{if}(\phi, \sigma_1, \sigma_2)$ in s corresponds to the execution of σ_1 if ϕ is true in s, else it corresponds to the execution of σ_2. A $\mathsf{while}(\phi, \sigma)$ loop may only result in a successor configuration if ϕ is true, in which case the remain program is $[\gamma, while(\phi, \sigma)]$, where again γ is what remains of σ. $(\sigma_1 \rangle\rangle \sigma_2)$ executes

σ_1 whenever possible; however, if σ_1 is blocked it executes σ_2.

The execution of the nondeterministic instruction $(\sigma_1|\sigma_2)$ in s can result in any successor configuration that can be reached through the execution of σ_1 or σ_2 in s. The execution of $\pi x.\sigma$ can result in a successor configuration $\langle \delta, s' \rangle$ if there is a v such that the execution of any σ_v^x can result in $\langle \delta, s' \rangle$. Here, σ_v^x is the program resulting from σ by substituting x with v. The execution of σ^* in s can result in a new situation s' if the execution of σ can do so. The remaining program then consists of the sequence $[\gamma, \sigma^*]$, where γ is what remains of σ after the transition from s to s'. Similarly, the execution of $\sigma^{||}$ in s can result in a new situation s' if the execution of σ can do so. This time, the remaining program consists of the concurrent execution of γ and $\sigma^{||}$. Finally, $(\sigma_1 \mid\mid \sigma_2)$ may result in any successor situation which can be reached by a single step of any σ_i. The remaining program then consists of the concurrent execution of what remains of σ_i, and the other (unmodified) process.

A final situation s' reachable after a finite number of transitions from a starting situation s is identified with the situation resulting from a possible execution trace of program σ, starting in situation s. This is captured by the predicate $Do(\sigma, s, s')$, which is defined in terms of $Trans^*$, the transitive closure of $Trans$:

$$Do(\sigma, s, s') \equiv \exists \delta. Trans^*(\sigma, s, \delta, s') \wedge Final(\delta, s')$$
$$Trans^*(\sigma, s, \delta, s') \equiv \forall T[... \supset T(\sigma, s, \delta, s')]$$

where the ellipsis stands for the universal closure of the conjunction of the following formulas:

$$T(\sigma, s, \sigma, s)$$
$$Trans(\sigma, s, \sigma^*, s^*) \wedge T(\sigma^*, s^*, \delta, s') \supset T(\sigma, s, \delta, s')$$

Given a program δ, proving that δ is executable in the initial situation then amounts to proving $\Gamma \models \exists s. Do(\delta, S_0, s)$, where Γ consists of the above axioms for ConGolog together with a situation calculus basic action theory. The following proposition (Lemma 4 from [dGLL00]) turns out to be quite useful in doing so. Intuitively, it says that to show that $Trans^*(\sigma, s, \delta, s')$ holds, it is sufficient to show that there is a sequence of transitions leading from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$.

Proposition 1: *Let* AX *be the foundational axioms of the situation calculus together with the definitions of Trans, Final and Trans*. Then for every model M of* AX, $M \models Trans^*(\sigma, s, \delta, s')$ *if and only if there exist* $\sigma_1, s_1, ..., \sigma_n, s_n$ *such that* $\sigma_1 = \sigma, s_1 = s, \sigma_n = \delta, s_n = s'$ *and* $M \models Trans(\sigma_i, s_i, \sigma_{i+1}, s_{i+1})$ *for* $i = 1, ...n-1$.

3.2.2 An Example

To illustrate the use of ConGolog, let us go back to the coffee delivery example of the previous section. Suppose we want to instruct the robot to subsequently serve coffee to all who have issued a coffee request. This can be specified by the following ConGolog plan:

$$\Pi_{deliv} \doteq \mathsf{while}(\exists r. CoffeeRequest(r), \\ [gotoRoom(nextRoom), giveCoffee(robotLoc)])$$

A word on notation: throughout this thesis, we will use terms of the form Π_{name} to denote plans. Note that we have not yet defined the semantics of procedures; accordingly we consider Π_{deliv} as an abbreviation. Π_{deliv} makes use of the defined functional fluent *nextRoom* to

determine the order in which coffee requests are to be fulfilled. *nextRoom* is defined as follows:

$$\begin{aligned} &\mathit{nextRoom}(s) = r \equiv r = \mathsf{nil} \wedge \forall r'.\neg \mathit{CoffeeRequest}(r', s) \vee \\ &\quad \mathit{CoffeeRequest}(r, s) \wedge \neg\exists r'.[\mathit{CoffeeRequest}(r', s) \wedge \mathit{roomOrder}(r', r)]. \end{aligned} \tag{3.12}$$

Here, the predicate *roomOrder* defines an (arbitrary) total ordering on the room names. For means of simplicity, we use the following definition of *roomOrder*:

$$\begin{aligned} \mathit{roomOrder}(r', r) \equiv & r' = R_1 \wedge r \in \{R_2, R_3, R_4\} \vee \\ & r' = R_2 \wedge r \in \{R_3, R_4\} \vee \\ & r' = R_3 \wedge r \in \{R_4\}. \end{aligned} \tag{3.13}$$

Let Γ be the basic action theory describing the coffee delivery domain from the previous section together with the axioms defining *Trans*, *Final*, *Trans** and *Do* from this section, the definitions of *nextRoom* and *roomOrder* and the axioms needed for the encoding of programs as first-order terms (cf. [dGLL00] and Appendix A). Then from Γ we can deduce that the execution of Π_{deliv} in S_0 results in a situation (an execution trace) where all requests have been satisfied:

$$\begin{aligned} \Gamma \models \exists s.\mathit{Do}(\Pi_{deliv}, S_0, s) \wedge \neg\exists r.\mathit{CoffeeRequest}(r, s) \wedge \\ s = \mathit{do}([\mathit{gotoRoom}(R_2), \mathit{giveCoffee}(R_2), \mathit{gotoRoom}(R_4), \mathit{giveCoffee}(R_4)], S_0). \end{aligned}$$

Proof: Although the proof is straightforward, it is quite laborious. By Proposition 1, to prove that $\mathit{do}([\mathit{gotoRoom}(R_2), \mathit{giveCoffee}(R_2), \mathit{gotoRoom}(R_4), \mathit{giveCoffee}(R_4)], S_0)$ is an execution trace of Π_{deliv} in S_0 we have to show that there is a sequence of transitions from S_0 to $\mathit{do}([\mathit{gotoRoom}(R_2), \mathit{giveCoffee}(R_2), \mathit{gotoRoom}(R_4), \mathit{giveCoffee}(R_4)], S_0)$. Furthermore, we have to show that the last configuration is final. From the initial state description (Axiom 3.7), we get:

$$\mathit{CoffeeRequest}(R_2, S_0). \tag{3.14}$$

Hence, $\exists r.\mathit{CoffeeRequest}(r, S_0)$. From this and the definition of *Final*:

$$\neg \mathit{Final}(\Pi_{deliv}, S_0). \tag{3.15}$$

From the definition of *Trans* regarding while-loops:[5]

$$\begin{aligned} &\mathit{Trans}(\mathsf{while}(\exists r.\mathit{CoffeeRequest}(r), \\ &\qquad [\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})]), S_0, \delta, s') \equiv \\ &\exists \gamma.\mathit{Trans}([\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})], S_0, \gamma, s') \wedge \\ &\exists r.\mathit{CoffeeRequest}(r, S_0) \wedge \\ &\delta = [\gamma, \mathsf{while}(\exists r.\mathit{CoffeeRequest}(r), \\ &\qquad [\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})])]. \end{aligned} \tag{3.16}$$

From the initial state description (Axiom 3.7), the definition of *nextRoom* (Axiom 3.12) and the definition of *roomOrder* (Axiom 3.13):

$$\mathit{nextRoom}(S_0) = R_2. \tag{3.17}$$

[5] To be more precise, we also need the predicate $\mathsf{Holds}(\phi, s)$, which is used to evaluate reified conditions used in tests ("$\phi[s]$"). See [dGLL00] and Appendix A.

From 3.17, the action precondition for *gotoRoom* (Axiom 3.8) and the definition of *Trans* regarding primitive actions:

$$Trans(gotoRoom(nextRoom), S_0, \delta, s') \equiv s' = do(gotoRoom(R_2), S_0) \wedge \delta = \mathsf{nil}. \tag{3.18}$$

From 3.18 and the definition of *Trans* regarding sequences:

$$\begin{array}{c} Trans([gotoRoom(nextRoom), giveCoffee(robotLoc)]), S_0, \delta, s') \equiv \\ s' = do(gotoRoom(R_2), S_0) \wedge \delta = [\mathsf{nil}, giveCoffee(robotLoc)]. \end{array} \tag{3.19}$$

From 3.14, 3.16 and 3.19 we get the following, which shows that there is a transition from S_0 to $do(gotoRoom(R_2), S_0)$.

$$\begin{array}{l} Trans(\mathsf{while}(\exists r. CoffeeRequest(r), \\ \qquad [gotoRoom(nextRoom), giveCoffee(robotLoc)]), S_0, \delta, s') \equiv \\ \delta = [[\mathsf{nil}, giveCoffee(robotLoc)], \mathsf{while}(\exists r. CoffeeRequest(r), \\ \qquad\qquad [gotoRoom(nextRoom), \\ \qquad\qquad giveCoffee(robotLoc)])] \wedge \\ s' = do(gotoRoom(R_2), S_0). \end{array} \tag{3.20}$$

Let $S_1 \doteq do(gotoRoom(R_2), S_0)$. From the definition of *Final*:

$$\neg Final([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_1). \tag{3.21}$$

From the definition of *Trans* regarding sequences and 3.21:

$$\begin{array}{l} Trans([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_1, \delta, s') \equiv \\ \exists \gamma. Trans([\mathsf{nil}, giveCoffee(robotLoc)], S_1, \gamma, s') \wedge \\ \delta = [\gamma, \Pi_{deliv}]. \end{array} \tag{3.22}$$

From the definition of *Trans* regarding sequences and the fact $Final(\mathsf{nil}, s)$:

$$Trans([\mathsf{nil}, giveCoffee(robotLoc)], s, \delta, s') \equiv Trans(giveCoffee(robotLoc), s, \delta, s'). \tag{3.23}$$

From the successor state axiom for *robotLoc* (Axiom 3.10):

$$robotLoc(S_1) = R_2. \tag{3.24}$$

From 3.24, the action precondition axiom for *giveCoffee* (Axiom 3.9) and the definition of *Trans* regarding primitive actions:

$$\begin{array}{c} Trans(giveCoffee(robotLoc), S_1, \delta, s') \equiv \\ s' = do(giveCoffee(R_2)], S_1) \wedge \delta = \mathsf{nil}. \end{array} \tag{3.25}$$

From 3.25 and 3.23 and 3.22 we get the following, which shows that there is a transition from S_1 to $do(giveCoffee(R_2), S_1)$:

$$\begin{array}{c} Trans([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_1, \delta, s') \equiv \\ \delta = [\mathsf{nil}, \Pi_{deliv}] \wedge s' = do(giveCoffee(R_2), S_1). \end{array} \tag{3.26}$$

Let $S_2 \doteq do(giveCoffee(R_2), S_1)$. From the successor state axiom for *CoffeeRequest* (Axiom 3.11) and the initial state description (Axiom 3.7):

$$CoffeeRequest(R_4, S_2). \tag{3.27}$$

From 3.27 and the definition of *Final*:

$$\neg Final([\mathsf{nil}, \Pi_{deliv}], S_2). \tag{3.28}$$

From the definition of *Trans* regarding sequences and the fact $Final(\mathsf{nil}, s)$:

$$\begin{array}{c} Trans([\mathsf{nil}, \Pi_{deliv}], S_2, \delta, s') \equiv \\ Trans(\Pi_{deliv}, S_2, \delta, s'). \end{array} \tag{3.29}$$

From 3.29, 3.27 and the definition of *Trans* regarding while-loops:

$$\begin{array}{l} Trans([\mathsf{nil}, \Pi_{deliv}], S_2, \delta, s') \equiv \\ \quad Trans([gotoRoom(nextRoom), giveCoffee(robotLoc)], S_2, \gamma, s') \wedge \\ \quad \delta = [\gamma, \Pi_{deliv}]. \end{array} \tag{3.30}$$

From the definition of *nextRoom* (Axiom 3.12), the successor state axiom for *CoffeeRequest* (Axiom 3.11) and the definition of *roomOrder* (Axiom 3.13):

$$nextRoom(S2) = R_4. \tag{3.31}$$

From 3.30, 3.31, the action precondition axiom for *gotoRoom* (Axiom 3.8) and the definition of *Trans* we get the following, which shows that there is a transition from S_2 to $do(gotoRoom(R_4), S_2)$:

$$\begin{array}{l} Trans([\mathsf{nil}, \Pi_{deliv}], S_2, \delta, s') \equiv \\ \quad s' = do(gotoRoom(R_4), S_2) \wedge \\ \quad \delta = [[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}]. \end{array} \tag{3.32}$$

Let $S_3 \doteq do(gotoRoom(R_4), S_2)$. From the definition of *Final*:

$$\neg Final([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_3) \tag{3.33}$$

From the successor state axiom for *robotLoc* (Axiom 3.10):

$$robotLoc(S_3) = R_4 \tag{3.34}$$

From the definition of *Trans* regarding sequences and 3.33:

$$\begin{array}{l} Trans([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_3, \delta, s') \equiv \\ \quad \exists \gamma. Trans([\mathsf{nil}, giveCoffee(robotLoc)], S_3, \gamma, s') \wedge \\ \quad \delta = [\gamma, \Pi_{deliv}]. \end{array} \tag{3.35}$$

From the action precondition axiom for *giveCoffee* (Axiom 3.9), 3.34 and the definition of *Trans*:

$$\begin{array}{c} Trans([\mathsf{nil}, giveCoffee(robotLoc)], S_3, \delta, s') \equiv \\ s' = do(giveCoffee(R_4), S_3) \wedge \delta = \mathsf{nil}. \end{array} \tag{3.36}$$

From 3.35 and 3.36, we get the following, which shows that there is a transition from S_3 to $do(giveCoffee(R_4), S_3)$:

$$\begin{array}{c} Trans([[\mathsf{nil}, giveCoffee(robotLoc)], \Pi_{deliv}], S_3, \delta, s') \equiv \\ s' = do(giveCoffee(R_4), S_3) \wedge \delta = [\mathsf{nil}, \Pi_{deliv}]. \end{array} \tag{3.37}$$

Let $S_4 \doteq do(giveCoffee(R_4), S_3)$. We have now shown that there is a sequence of transitions from S_0 to $S_4 = do([gotoRoom(R_2), giveCoffee(R_2), gotoRoom(R_4), giveCoffee(R_4)], S_0)$. From 3.20, 3.26, 3.32, 3.37 and Proposition 1:

$$Trans^*(\Pi_{deliv}, S_0, [\mathsf{nil}, \Pi_{deliv}], S_4). \tag{3.38}$$

In order to show that S_4 is an execution trace of Π_{deliv}, we still have to show that the configuration reached is final. From the successor state axiom for *CoffeeRequest* (Axiom 3.11) and the initial state description (Axiom 3.7):

$$\neg\exists r. CoffeeRequest(r, S_4). \tag{3.39}$$

From 3.39 and the definition of *Final* regarding while-loops:

$$\begin{aligned} Final(&\mathsf{while}(\exists r. CoffeeRequest(r), \\ &[gotoRoom(nextRoom), giveCoffee(robotLoc)]), S_4). \end{aligned} \tag{3.40}$$

From 3.40, the fact $Final(\mathsf{nil}, s)$ and the definition of *Final* regarding sequences:

$$Final([\mathsf{nil}, \Pi_{deliv}], S_4). \tag{3.41}$$

Finally, from the definition of *Do*, 3.38 and 3.41, we can conclude that S_4 is an execution trace of Π_{deliv} in S_0:

$$\Gamma \models Do(\Pi_{deliv}, S_0, S_4). \tag{3.42}$$

This, together with 3.39, finishes the proof. □

As the above proof illustrates, proving that the execution of a program results in a particular execution trace is a straightforward but laborious task. In general, it is suggestive to make use of a ConGolog implementation instead of calculating execution traces by hand. In fact, all example executions presented in this thesis were computed using the prototypical implementations described in Chapter 8. Accordingly, in the remainder of this thesis we will only provide some (informal) arguments as to why a program results in a particular execution trace.

3.2.3 Extending the Transition Semantics to Procedures

This subsection describes how the transition semantics of ConGolog can be extended to deal with *procedures*. Extending the semantics to deal with procedures necessitates the use of a second-order definition of *Trans* and *Final*, because a recursive procedure may do an *arbitrary* number of procedure calls before it performs a primitive action or test, and such procedure calls are not viewed as transitions. The material presented in this subsection is quite technical and can be skipped by the reader on a first reading.

A central notion in the definition of the semantics of ConGolog regarding procedures is the concept of an *environment*. An environment $\mathcal{E}$ is a collection of procedure definitions $\mathsf{proc}(P_1(\vec{v_1}), \beta_1); ...; \mathsf{proc}(P_n(\vec{v_n}), \beta_n)$, where P_i is the name of the i-th procedure in $\mathcal{E}$, $\vec{v_i}$ are its formal parameters and β_i is its procedure body. A procedure body is a ConGolog program, possibly including both procedure calls and new procedure definitions.

Formally, to extend the semantics of ConGolog to deal with procedures we have to consider three new program constructs:

- $\{\mathcal{E};\sigma\}$, where $\mathcal{E}$ is an environment and σ a program extended with procedure calls. This is the $\{\mathsf{proc}(P_1(\vec{v_1}),\beta_1);...;\mathsf{proc}(P_n(\vec{v_n}),\beta_n);\sigma\}$ construct already listed in Section 3.2. $\{\mathcal{E};\sigma\}$ binds procedure calls in σ to the definitions given in $\mathcal{E}$. The usual notion of free and bound apply, so for example in $\{\mathsf{proc}(P_1(),a);[P_2,P_1]\}$, P_1 is bound but P_2 is free.
- $P(\vec{t})$, where P is a procedure name and $\vec{t}$ actual parameters associated to the procedure P; as usual, the situation argument in the terms constituting $\vec{t}$ is replaced by *now*. $P(\vec{t})$ denotes a procedure call, which invokes procedure P on the actual parameters $\vec{t}$ evaluated in the current situation.
- $[\mathcal{E} : P(\vec{t})]$, where $\mathcal{E}$ is an environment, P a procedure name and $\vec{t}$ actual parameters associated to the procedure P. $[\mathcal{E} : P(\vec{t})]$ denotes a procedure call that has been *contextualized* by the environment in which the definition of P is to be looked for is $\mathcal{E}$.

The semantics of ConGolog programs with procedures is specified by providing a second-order definition for both *Trans* and *Final*. *Trans* is defined as follows:

$$Trans(\sigma,s,\delta,s') \equiv \forall T.[... \supset T(\sigma,s,\delta,s')]. \tag{3.43}$$

Here, the ellipsis stands for the universal closure of the conjunction of the original set of axioms for *Trans* modulo textual substitution of *Trans* with T, together with the following two assertions:

$$T(\{\mathcal{E};\sigma\},s,\delta,s') \equiv T(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]},s,\delta,s'); \tag{3.44}$$

$$T([\mathcal{E} : P(\vec{t})],s,\delta,s') \equiv T(\{\mathcal{E};\beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\},s,\delta,s'). \tag{3.45}$$

Here, $\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}$ denotes the program σ with all procedures bound by $\mathcal{E}$ and free in σ replaced by their contextualized version, and where $\beta_P{}^{\vec{v_p}}_{\vec{t[s]}}$ denotes the body of the procedure P in $\mathcal{E}$ with formal parameters $\vec{v_p}$ substituted by the actual parameters $\vec{t}$ evaluated in the current situation. The first of these two axioms says that when a program with an associated environment $\mathcal{E}$ is executed, all procedure calls bound by $\mathcal{E}$ are simultaneously substituted by procedure calls contextualized by the environment of the procedure. The second axiom says that when a contextualized procedure call is executed, the call is replaced by the body of the procedure, associated with $\mathcal{E}$ in order to deal with further procedure calls according to the *lexical* (or *static*) scoping rule. Furthermore, the actual parameters are evaluated in the current situation, and then are substituted for the formal parameters in the procedure bodies, which yields *call-by-value* parameter passing.

Similarly, *Final* is defined as follows:

$$Final(\sigma,s) \equiv \forall F.[... \supset F(\sigma,s)],$$

where the ellipsis stands for the universal closure of the conjunction of the original set of axioms for *Final* modulo textual substitution of *Final* with F, together with the following assertions:

$$F(\{\mathcal{E};\sigma\},s) \equiv F(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]},s);$$
$$F([\mathcal{E} : P(\vec{t})],s) \equiv F(\{\mathcal{E};\beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\},s).$$

In [dGLL00], de Giacomo, Lespérance and Levesque show the following equivalence (their Theorem 4):

Proposition 2: *With respect to programs without procedures, the second order definition of Trans and Final introduced above is equivalent to the versions introduced in Section 3.2.1.*

Note that no assertions for uncontextualized procedure calls are present in the definitions of *Trans* and *Final*: a procedure call which cannot be bound to a procedure definition neither can do transitions nor can be considered successfully completed. In particular, the second-order definition allows to assign a formal semantics to every recursive procedure, including vicious circular ones. The definition of *Trans* disallows the (step-wise) execution of such ill-formed procedures, and at the same time through the definition of *Final* they are not considered as completed. As an example, let us consider the ill-formed program $\{\mathsf{proc}(P(), P()); P()\}$. We will use $\mathcal{E}_1$ as an abbreviation for $\mathsf{proc}(P(), P())$. The program $\{\mathcal{E}_1; P()\}$ can do a transition if and only if $\mathit{Trans}(\{\mathcal{E}_1; P()\}, s, \delta, s')$ holds for some $\langle s', \delta\rangle$. From the second-order definition of *Trans* and the fact that $\{\mathcal{E}_1; P()\}$ does only involve procedure definition and invocation follows that $\mathit{Trans}(\{\mathcal{E}_1; P()\}, s, \delta, s')$ holds if and only if $T(\{\mathcal{E}_1; P()\}, s, \delta, s')$ holds for all predicates T that satisfy the two assertions 3.44 and 3.45. This is equivalent to the following:

$$\begin{gathered} T(\{\mathcal{E}_1; P\}, s, \delta, s') \equiv \\ T([\mathcal{E}_1 : P], s, \delta, s') \equiv \\ T(\{\mathcal{E}_1; P\}, s, \delta, s'). \end{gathered}$$

The above does not cause any implications regarding the truth value of $T(\{\mathcal{E}_1; P\}, s, \delta, s')$. As a result, the second-order definition of *Trans* yields $\mathit{Trans}(\{\mathsf{proc}(P(), P()); P()\}, s, \delta, s') \equiv$ FALSE. Similarly, it is possible to verify that $\{\mathsf{proc}(P(), P()); P()\}, s, \delta, s')\}$ is not final. Thus, $\{\mathsf{proc}(P(), P()); P()\}$ can neither perform transitions nor is it final.

On the other hand, if we consider programs with well-defined procedures (or without procedures), we obtain clear implications for the predicates T. As an example, let us consider a modification of the coffee delivery example, where the robot plan is specified by a *procedure*. Let

$$\begin{aligned} \mathcal{E}_2 \doteq \mathsf{proc}(&\Pi_{deliv}(),\mathsf{while}(\exists r.\mathit{CoffeeRequest}(r), \\ &[\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})]). \end{aligned}$$

Then, by the new definition of *Trans* we get:

$$\begin{aligned} &\mathit{Trans}(\{\mathcal{E}_2; \Pi_{deliv}()\}, s, \delta, s') \equiv \\ &\quad \mathit{Trans}([\mathcal{E}_2 : \Pi_{deliv}()], s, \delta, s') \equiv \\ &\quad \mathit{Trans}(\{\mathcal{E}_2; \mathsf{while}(\exists r.\mathit{CoffeeRequest}(r), \\ &\qquad\qquad [\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})])\}, s, \delta, s') \equiv \\ &\quad \mathit{Trans}(\mathsf{while}(\exists r.\mathit{CoffeeRequest}(r), \\ &\qquad\qquad [\mathit{gotoRoom}(\mathit{nextRoom}), \mathit{giveCoffee}(\mathit{robotLoc})]), s, \delta, s'). \end{aligned}$$

Thus, we can deduce that the execution of $\{\mathcal{E}_2; \Pi_{deliv}\}$ in S_0 corresponds to the execution of the procedure body of Π_{deliv}. From this and the considerations in Section 3.2.2, we can then conclude:

$$\begin{aligned} \Gamma \models \exists s.&\mathit{Do}(\{\mathcal{E}_2; \Pi_{deliv}\}, S_0, s) \wedge \\ &s = \mathit{do}([\mathit{gotoRoom}(R_2), \mathit{giveCoffee}(R_2), \mathit{gotoRoom}(R_4), \mathit{giveCoffee}(R_4)], S_0). \end{aligned}$$

For simplicity, in the remainder of this thesis we use the convention to first present the definition of procedures in the form $\mathsf{proc}(\Pi_{name}, body)$, and thereafter to simply use Π_{name} as a shorthand for the program $\{\mathsf{proc}(\Pi_{name}, body); \Pi_{name}\}$.

3.3 A Probabilistic, Epistemic Situation Calculus

The situation calculus presented in Section 3.1 allows us to talk only about how the actual world evolves, starting in the initial situation S_0. But in many realistic scenarios, a robot or agent is uncertain about the (initial) state of the world. For example, a robot might not know whether a door is open or not, or might be uncertain about its actual position. Instead, the robot often only has probabilistic knowledge, like "the door is open with probability 70%". To take this into account, Bacchus, Halpern and Levesque (BHL) [BHL95, BHL99] propose to characterize an agent's epistemic state in a probabilistic fashion. More specifically, they propose to characterize an epistemic state by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities. We will now describe in somewhat more detail how to specify and reason about a probabilistic epistemic state in the situation calculus.

Formally, there is a binary functional fluent $p(s', s)$ which can be read as "in situation s, the agent thinks that s' is possible with degree of likelihood $p(s', s)$." All weights must be non-negative and situations considered impossible will be given weight 0:

$$S_0 \prec s \supset p(s', s) \geq 0.$$

We do not require that the degrees of likelihood (which we also call "weights") of all possible situations sum to 1, that is p represents an *unnormalized* probability distribution. However, we assume that the sum of the weights is finite in order to assure that the unnormalized probability distribution is well-defined:

$$S_0 \prec s \supset \exists n. \sum_{s'} p(s', s) = n$$

Appendix B describes how this and similar summations can be expressed using second-order quantification. Finally, all situations considered possible in S_0 must be *initial*. Here, we write $Init(\sigma)$ to say that σ is an initial situation. Formally, $Init(\sigma)$ is an abbreviation defined as follows:

$$Init(\sigma) \doteq \forall s, a. \sigma \neq do(a, s).$$

3.3.1 Foundational Axioms for the Epistemic Situation Calculus

Having more than one initial situation means that Reiter's induction axiom for situations (foundational axiom 3) no longer holds, just as in [BHL99]. Instead, we consider the following set of foundational axioms, adapted from [LPR98, LL98]:

1. $Init(S_0) \wedge \forall s'. p(s', S_0) > 0 \supset Init(s')$;
2. $do(a, s) = do(a', s') \supset a = a' \wedge s = s'$;
3. The following is a weaker version of the original induction axiom:
   $\forall P. \{\forall s. Init(s) \supset P(s) \wedge [\forall s, a. P(s) \supset P(do(a, s)))]\} \supset \forall s P(s)$;

4. $Init(s') \supset \neg(s \sqsubset s')$;

5. $s \sqsubset do(a, s') \equiv s \sqsubseteq s'$, where $s \sqsubseteq s'$ stands for $(s \sqsubset s') \vee (s = s')$;

6. $s \prec s' \equiv s \sqsubset s' \wedge \forall a, s^*.s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*)$.

The first three foundational axioms characterize the space of all situations. Unlike the foundational axioms for the non-epistemic situation calculus, which characterize the space of all situations as a tree with root S_0, the foundational axioms for the epistemic situation calculus make the set of situations isomorphic to a set of trees (a forest), where the roots of the different trees are the initial situations. The last three axioms characterize the binary predicates $\sqsubset$ and $\prec$, just as in Section 3.1.2.

3.3.2 Belief

Based on p, BHL define $Bel(\phi, s)$, the agent's degree of belief that ϕ holds in situation s. As usual, ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation. Formally, they define $Bel(\phi, s) = p$ as an abbreviation for the following term expressible in second-order logic:

$$\sum_{s':\phi[s']} p(s', s) / \sum_{s'} p(s', s) = p.$$

Intuitively, $Bel(\phi, s)$ is the (normalized) sum of the degree of likelihood of all situations s' considered possible in s that fulfill ϕ. Note that we are restricting ourselves to discrete probability distributions, where the probability of a set can be computed as the sum of the probabilities of the elements of the set.

As an example, let us describe the initial belief state of a robot whose confidence to be in room R_1 is 90%, and which believes to be in room R_2 else. In this situation, the world is initially in one of two states, $s1$ and $s2$, which occur with probability 0.9 and 0.1, respectively. In this simple scenario, these are the only possibilities, all other situations have likelihood 0. The following axiom makes this precise together with what holds and does not hold in each of the two states:

$$\begin{aligned}\exists s_1, s_2.\forall s.s \neq s_1 \wedge s \neq s_2 \supset p(s, S_0) = 0 \wedge \\ p(s_1, S_0) = 9 \wedge p(s_2, S_0) = 1 \wedge \\ robotLoc(s_1) = R_1 \wedge robotLoc(s_2) = R_2.\end{aligned}$$

From this, it is easy to deduce $Bel(robotLoc(now) = R_1, S_0) = 0.9$.

Chapter 4

cc-Golog – Dealing with Continuous Change

In real-world applications, high-level controllers typically operate low-level processes which change the world in a continuous fashion over time. Although it may sometimes seem reasonable to abstract away from the continuous change involved, for example to represent a pickup as causing discrete qualitative change, abstracting away from the actual continuous movement of the gripper, this is not always appropriate. For example, consider a robot operating in the north floor of the Computer Science Department V at Aachen University of Technology, illustrated in Figure 4.1. Here, dealing with event-driven actions like "say hello if you come near Door 6213" is both problematic and un-natural without taking into account the continuous trajectory caused by the navigation process.

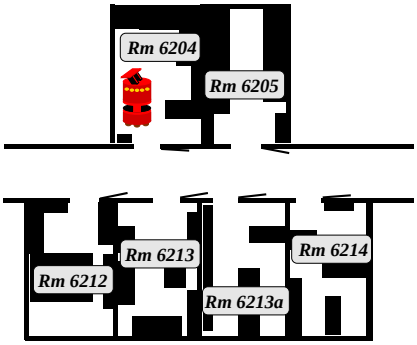


Figure 4.1: Robot environment

In order to deal with continuously changing properties in a natural way, the non-logic-based robot programming languages RPL, RAP, PRS-Lite and Colbert introduced in Section 2.3 all offer a special wait-for instruction, which suspends activity until a condition appealing to continuously changing properties becomes true. As an example, consider the following RPL program:

```
(with-policy (loop
                wait-for Batt-Level ≤ 46
                charge-batteries)
  (with-policy (loop
                  wait-for near-door(Rm 6213)
                  say("hello")
                  wait-for ¬near-door(Rm 6313))
    deliver-mail))
```

Figure 4.2: Office delivery plan

Roughly, the robot's main task is to deliver mail, which we merely indicate by a call to

the procedure deliver-mail. While executing this procedure, the robot concurrently also does the following, with an increasing level of priority: whenever it passes the door to Room 6213 it says "hello" and, should the battery level drop dangerously low, it recharges its batteries interrupting whatever else it is doing at this moment. Even this simple program exhibits important features of high-level robot controllers:

- The timing of actions is largely *event-driven*, that is, rather than explicitly stating when an action occurs, the execution time depends on certain conditions becoming true such as reaching a certain door. In this example, the specification of reactive behavior is realized using the wait-for construct.
- Actions are executed *as soon as possible*, that is, immediately after their precondition becoming true. For example, the batteries are charged immediately after a low voltage level is determined.
- Conditions such as the voltage level are best thought of as changing *continuously* over time.
- Parts of programs which execute with high priority must allow the execution of concurrent sub-programs with lower priority, other parts must interdict the execution of concurrent sub-programs with lower priority. For example, while waiting for a low battery level, mail delivery should continue. On the other hand, the actual charging of the battery should block the mail delivery. We will refer to high-priority sub-programs that interdict the execution of concurrent programs as *blocking policies*. In RPL, blocking policies are handled by having the different procedures (for example deliver-mail and charge-batteries) compete for a valve (cf. the discussion on page 34).

Given the inherent complexity of concurrent robot programs, answers to questions like whether a program is executable and whether it will satisfy the intended goals are not easy to come by, yet important to both the designer during program development and the robot who may want to choose among different courses of action. A principled approach to answering such questions is to *project* how the world evolves when actions are performed. Although the XFRM system [McD92a, McD94] allows the projection of RPL plans like the above [BG98], it relies on RPL's run-time system, which lacks a formal semantics and which makes predictions implementation dependent. Preferably one would like a language which is as powerful as RPL yet allows for projections based on a perspicuous, declarative semantics. As for other non-logic-based robot control languages discussed in Section 2.3, they either model complex programs as primitive actions (like RAP), or do not consider projection at all.

Although ConGolog offers features such as concurrency and priorities, it turns out that in its current form it is not suitable to represent robot controllers such as the example above. The problem is that in its current form, the situation calculus on which it is based is only able to represent *discrete* change, and lacks the capability to express that a property like the robot's position changes *continuously* over time. On the other hand, the existing temporal extension of GOLOG [Rei98] requires that the execution time of an action is supplied *explicitly*, which seems incompatible with event-driven specifications. Finally, it is not clear how to handle both blocking and non-blocking prioritized execution in ConGolog. In particular, ConGolog's $(\sigma_1 \rangle\rangle \sigma_2)$ construct simply blocks the execution of of σ_2 whenever σ_1 can execute (cf. page 43).

In this chapter, we will present an extension of GOLOG and the situation calculus which will allow the specification and projection of plans which react to continuously changing

properties, similarly to the example of Figure 4.2. We proceed in three steps. First we present a new extension of the situation calculus which, besides dealing with continuous change, allows us to model actions which are event-driven by including a special action *waitFor* in the logic. We then turn to a new variant of ConGolog called cc-Golog, which is based on the extended situation calculus. In particular, cc-Golog comes with a new semantics for concurrent execution, which ensures that actions which can be executed earlier are always preferred. Unlike ConGolog, cc-Golog only provides *deterministic instructions*, which is due to anomalies which arise from the interaction of *waitFor*-actions, concurrency and nondeterminism.

Finally, we show how a robot control architecture where a high-level controller communicates with low-level processes via messages (like the layered robot control architectures discussed in Section 2.3) can be modeled directly in cc-Golog. Thereby, we model complex low-level processes like the navigation process *as cc-Golog programs*. This allows a very fine-grained characterization of the effects of the low-level processes at a level of detail involving many atomic actions, in particular taking into account the temporal extent of low-level processes like the navigation process. Given a faithful characterization of the low-level processes in terms of cc-Golog programs, we can then reason about the effects of their activation through simulation of the corresponding cc-Golog model. The main advantage of having an explicit model of the robot control architecture is that there is a clear separation of the actions of the high-level controller from those of low-level processes. We remark that in this chapter, we are only concerned with how to project a cc-Golog plan. Actual (on-line) execution of cc-Golog plans will be the topic of the next chapter.

The rest of this chapter is organized as follows. In the next section, we show how to extend the basic situation calculus to include continuous change and time. Thereafter, we present cc-Golog which takes into account the extended situation calculus, in particular its notion of time and temporal precedence. Finally, we show how to model a layered robot control architecture, develop a cc-Golog model of a low-level navigation process, and show how the example-program can be specified quite naturally in cc-Golog with the additional benefit of supporting projections firmly grounded in logic.

4.1 Continuous Change and Time

Actions in the situation calculus cause discrete changes and, in its basic form, there is no notion of time. In robotics applications, however, we are faced with processes such as navigation which cause properties like the robot's location and orientation to change continuously over time. In order to model such processes in the situation calculus in a natural way, we add continuous change and time directly to its ontology.

4.1.1 Adding a Timeline

As demonstrated by Miller, Pinto and Reiter [Mil96, Pin97, Rei96], adding time is a simple matter. We add a new sort *Real* ranging over the real numbers and, for mnemonic reasons, another sort *Time* ranging over the reals as well. For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations; confer [Har96] for an in-depth treatment of theorem proving with reals. In order to connect situations and time, we add a special unary functional fluent *start* to the language with the understanding that $start(s)$ denotes the time when situation s begins. We will see later how *start* obtains its values and, in particular, how the passage of time is modeled.

4.1.2 Continuous Fluents

A fundamental assumption of the situation calculus is that fluents have a fixed value at every given situation. In order to see that this assumption still allows us to model continuous change, let us consider the example of a mobile robot moving along a straight line in a 1-dimensional world, that is, the robot's location at any given time is simply a real number. There are two types of actions the robot can perform, *startGo*(v), which initiates moving the robot with speed v, and *endGo* which stops the movement of the robot. The robot can start and stop moving along the line at any time.

Let us denote the robot's location by the fluent *robotLoc1d*. What should the value of *robotLoc1d* be after executing *startGo*(v) in situation s? Certainly it cannot be a fixed real value, since the position should change over time as long as the robot moves. In fact, the location of the robot at any time after *startGo*(v) (and before the robot changes its velocity) can be characterized (in a somewhat idealized fashion) by the *function* $x_0 + v \times (t - t_0)$, where x_0 is the starting position and t_0 the starting time. The solution is then to take this *function of time* to be the value of *robotLoc1d*. We call functional fluents whose values range over functions of time *continuous fluents*. In the next subsection, we will see how functions of time can be integrated into the situation calculus.

4.1.3 Functions of Time

In order to represent the value of continuous fluents, we introduce a new sort *t-function*, whose elements are meant to be functions of time. Formally, then, a continuous fluent is a fluent of sort *t-function*. For our 1-dimensional robot example, it suffices to consider two kinds of *t-function*s: constant functions, denoted by *constant*(x) and the special linear functions introduced above, which we denote as *linear*(x, v, t_0). In order to say what values these functions have at any particular time t, we use a new binary function *val*. In the example, we would add the following axioms:

$$val(constant(x), t) = x;$$
$$val(linear(x_0, v, t_0), t) = x_0 + v \times (t - t_0).$$

4.1.4 The passage of Time

Let us now turn to the issue of modeling the passage of time during a course of actions. As indicated in the introduction of this chapter, motivated by the treatment of time in robot control languages like RPL, RAP, or Colbert, we introduce a new type of primitive action *waitFor*(τ). The intuition is as follows: normally, every action happens immediately, that is, the starting time of the situation after doing a in s is the same as the starting time of s. The only exception is *waitFor*(τ): whenever this action occurs, the starting time of the resulting situation is advanced to the earliest time in the future when τ becomes true. Note that this has the effect that actions always happen as soon as possible.

One may object that requiring that two actions other than *waitFor* must happen at the same time is unrealistic. However, in robotics applications, actions often involve little more than sending messages in order to initiate or terminate processes so that the actual duration of such actions is negligible. For example, a procedure like mail delivery essentially consists of *startGo* and *waitFor* actions, which cause the robot to start moving towards its destination respectively to wait until it has been reached. In this example, a considerable amount of

time passes only while the high-level controller is waiting (i.e. the robot is traveling to its destination). Moreover, if two actions cannot happen at the same time, they can always be separated explicitly using *waitFor*.

The arguments of *waitFor* are restricted to what we call a *t-form*, which is a Boolean combination of closed atomic formulas of the form $(F\ op\ r)$, where F is a continuous fluent where *now* may refer to the actual situation, $op \in \{<, =\}$, and r is a term of type real (not mentioning *val*). We freely use $\leq$, $\geq$, or $>$ as well. An example is $\tau = (robotLoc1d \geq 1000)$. To evaluate a *t-form* at a situation s and time t, we write $\tau[s,t]$ which results in a formula which is like τ except that every continuous fluent F is replaced by $val(F(s),t)$. For instance, $(robotLoc1d \geq 1000)[s,t]$ becomes $(val(robotLoc1d(s),t) \geq 1000)$. We will not go into the details of reifying *t-form*s within the language; a thorough treatment of these issues can be found in Appendix A.1.4.

To see how actions are forced to happen as soon as possible, let $ltp(\tau, s, t)$ be an abbreviation for the formula

$$\tau[s,t] \wedge t \geq start(s) \wedge \forall t'.start(s) \leq t' \ < t \supset \neg\tau[s,t'],$$

that is, t in $ltp(\tau, s, t)$ is the least time point after the start of s where τ becomes true.[1] Then we require that a *waitFor*-action is possible if and only if the condition has a least time point. In practice, this means that it is up to the user to ensure that τ has a least time point.

$$Poss(waitFor(\tau), s) \equiv \exists t.ltp(\tau, s, t).$$

Intuitively, thus, a $waitFor(\tau)$ action is not possible in s if there is no least time point after the start of s where τ becomes true. For example, if $robotLoc1d(S_0) = linear(0, 1, start(S_0))$, then $waitFor(robotLoc > 1)$ is not possible in S_0 because there is no least time point t such that $0 + 1 \times t > 1$. On the other hand, it is straightforward to show that, if $\exists t.ltp(\tau, s, t)$ is satisfied, then t is unique.

Finally, we need to characterize how the fluent *start* changes its value when an action occurs. The following successor state axiom for *start* captures the intuition that the starting time of a situation changes only as a result of a $waitFor(\tau)$, in which case it advances to the earliest time in the future when τ holds.

$$\begin{aligned} Poss(a, s) \supset [start(do(a, s)) = t \equiv \\ \exists \tau.a = waitFor(\tau) \wedge ltp(\tau, s, t) \vee \forall \tau.a \neq waitFor(\tau) \wedge t = start(s)]] \end{aligned} \tag{4.1}$$

Let $\mathrm{AX}_{\mathrm{cc}}$ be the foundational axioms of the situation calculus from Section 3.1 together with the axioms required for *t-form*'s, the precondition axiom for *waitFor*, and the successor state axiom for *start*. Then, the following formulas are logical consequences of $\mathrm{AX}_{\mathrm{cc}}$.

Proposition 3:

1. *Actions happen as soon as possible:*
   $s \prec do(a, s) \supset [start(do(a, s)) = start(s)] \vee [\exists \tau.a = waitFor(\tau) \wedge ltp(\tau, s, start(do(a, s)))]$;

2. *The starting time of legal action sequences is monotonically nondecreasing:*
   $s \prec s' \supset start(s) \leq start(s')$.

[1] This is not unlike Reiter's definition of a least natural time point in the context of natural actions [Rei96]. Similar ideas occur in the context of delaying processes in real-time programming languages like Ada [BW91].

Proof: (1) Follows directly from Axiom 4.1 and the fact that a is possible in s.

(2) Follows by induction on n, the number of actions performed to obtain s' from s. $n = 1$ follows from (1), and the fact that, by definition, $ltp(\tau, s, t) \supset t \geq start(s)$. If $n > 1$, then $\exists a^*, s^*.s' = do(a^*, s^*) \wedge s \prec s^* \wedge Poss(a^*, s^*)$. By induction hypothesis, $start(s) \leq start(s^*)$. From this, together with (1) and $ltp(\tau, s, t) \supset t \geq start(s)$, we get $start(s) \leq start(s')$. □

To illustrate the approach, let us go back to the 1-dimensional robot example. First, we can formulate a successor state axiom for *robotLoc1d*:

$$\begin{aligned} Poss(a, s) \supset [&robotLoc1d(do(a, s)) = y \equiv \\ &\exists t_0, v, x.t_0 = start(s) \wedge x = val(robotLoc1d(s), t_0) \wedge \\ &\quad [a = startGo(v) \wedge y = linear(x, v, t_0) \vee \\ &\quad\ a = endGo \wedge y = constant(x)] \vee \\ &\neg\exists v.(a = startGo(v) \vee a = endGo) \wedge y = robotLoc1d(s)] \end{aligned}$$

In other words, when an action is performed, *robotLoc1d* is assigned either the function $linear(x, v, t_0)$, if the robot starts moving with velocity v and x is the location of the robot at situation s, or it is assigned $constant(x)$ if the robot stops, or it remains the same as in s. To see that the above successor state axiom is well-defined, let

$$\begin{aligned} \gamma_f(a, s, y) \doteq\ &\exists t_0, v, x.t_0 = start(s) \wedge x = val(robotLoc1d(s), t_0) \wedge \\ &[a = startGo(v) \wedge y = linear(x, v, t_0) \vee \\ &\ a = endGo \wedge y = constant(x)]. \end{aligned}$$

It is easy to see that

$$\forall a, s.[\exists y.\gamma_f(a, s, y) \equiv \exists v.(a = startGo(v) \vee a = endGo)].$$

Furthermore, for every *startGo* or *endGo* action the value of y is uniquely determined. Thus, from the unique names axioms for primitive actions, we get:

$$\neg\exists a, s, y, y'.\gamma_f(a, s, y) \wedge \gamma_f(a, s, y') \wedge y \neq y'$$

and hence the successor state axiom for $robotLoc1d(s)$ is equivalent to a successor state axiom of the form of Axiom 3.5 on page 41, and is well-defined (cf. Axiom 3.6).

Let Γ be $\mathrm{AX}_{\mathrm{cc}}$ together with the axioms for *val*, the successor state axiom for *robotLoc1d*, precondition axioms stating that *startGo* and *endGo* are always possible, and the facts $robotLoc1d(S_0) = constant(0)$ and $start(S_0) = 0$, that is, the robot initially rests at position 0 and the starting time of the initial situation is 0. Let us assume the robot starts moving at speed 50 (cm/s) and then waits until it reaches location 1000 (cm), at which point it stops. The resulting situation is $S_1 = do([startGo(50), waitFor(robotLoc1d = 1000), endGo], S_0)$. Then:

$$\Gamma \models start(S_1) = 20 \wedge robotLoc1d(S_1) = constant(1000).$$

Proof: (Sketch) In situation $do(startGo(50), S_0)$, the value of *robotLoc1d* is $linear(0, 50, 0)$. $val(linear(0, 50, 0), 20) = 1000$, and hence the least time point of the *t-form* $robotLoc1d = 1000$ in situation $do(startGo(50), S_0)$ is 20. Thus, in $do([startGo(50), waitFor(robotLoc1d = 1000)], S_0)$ the value of *start* is 20, and the value of *robotLoc1d* remains $linear(0, 50, 0)$. Finally, the execution of *endGo* leaves *start* unaffected (i.e. 20), and changes *robotLoc1d* to $constant(1000)$ because $val(linear(0, 50, 0), 20) = 1000$. □

In other words, the robot moves for 20 seconds and stops at location 1000, as one would expect.

4.1.5 A Simple Model of Robot Navigation

We will now model how in our introductory robot delivery example the robot's location changes as a result of primitive actions. For simplicity, we ignore the robot's orientation and represent its position by a tuple of reals. We begin by defining a 2-dimensional version of the *t-function*s *constant* and *linear*, ignoring the details of representing tuples of reals in logic:[2]

$$\begin{aligned}&val(constant(x,y),t) = \langle x,y\rangle\\ &val(linear(x,y,v_x,v_y,t_0),t) = \langle x',y'\rangle \equiv\\ &\quad x' = x + v_x * (t-t_0) \wedge y' = y + v_y * (t-t_0).\end{aligned}$$

While $constant(x,y)$ always evaluates to $\langle x,y\rangle$, $linear(x,y,v_x,v_y,t_0)$ is a linear function intended to approximate the movement of a robot in 2-dimensional space.

We represent the robot's position by the continuous fluent *robotLoc*, which evaluates to a 2-dimensional *t-function*. Again, *robotLoc* is affected by *startGo* and *endGo*, however this time *startGo* has two arguments. $startGo(\langle x,y\rangle)$ initiates moving the robot towards position $\langle x,y\rangle$ and *endGo* stops the movement of the robot. For simplicity, we model the robot as always traveling at speed 1m/s (in a more realistic model, we would also consider different velocities). Then, we obtain the following successor state axiom for *robotLoc*:

$$\begin{aligned}Poss(a,s) \supset [&robotLoc(do(a,s)) = f \equiv\\ &\exists t,x,y.\ t = start(s) \wedge val(robotLoc(s),t) = \langle x,y\rangle \wedge\\ &\quad [\exists x',y',v_x,v_y. a = startGo(\langle x',y'\rangle) \wedge\\ &\qquad v_x = (x'-x)/\nu \wedge v_y = (y'-y)/\nu \wedge f = linear(x,y,v_x,v_y,t) \vee\\ &\quad a = endGo \wedge f = constant(x,y) \vee\\ &\forall x',y'. a \neq startGo(x',y') \wedge a \neq endGo \wedge f = robotLoc(s)]]\end{aligned}$$

The variables x and y refer to the coordinates of the robot. After $startGo(\langle x',y'\rangle)$, *robotLoc* has as value a *linear t-function* starting at the current position and moving towards $\langle x',y'\rangle$. After *endGo*, it is $constant(\langle x,y\rangle)$. If a is neither a *startGo* nor a *endGo* action, *robotLoc* remains unchanged. Finally, $\nu \doteq \sqrt{(x'-x)^2 + (y'-y)^2}$ is a normalizing factor which ensures that the total 2-dimensional velocity is 1. In order to avoid $\nu = 0$, we assume that *startGo* is only possible if the destination differs from the robot's current location. Furthermore, we assume a battery level of at least 45 V. *endGo* is always possible. In the following axioms, t refers to the actual time, and x and y to the robot's current position:

$$\begin{aligned}&Poss(startGo(\langle x',y'\rangle),s) \equiv\\ &\quad \exists t,x,y.\ t = start(s) \wedge val(robotLoc(s),t) = \langle x,y\rangle \wedge \langle x,y\rangle \neq \langle x',y'\rangle \wedge\\ &\quad battLevel[s,t] \geq 45;\\ &\\ &Poss(endGo) \equiv \textsc{True}.\end{aligned}$$

Besides affecting the robot's position, *startGo* and *endGo* also have an effect on the robot's battery level. We assume that the battery level only decreases if the robot is moving, in which case it decreases with Δ_V V per second. The following successor state axiom for *battLevel* makes this precise:

[2] One possibility would be to treat $\langle x,y\rangle$ as an abbreviation for the term $coord(x,y)$ and to use unique names axioms for *coord*.

$$\begin{aligned} Poss(a,s) \supset [&battLevel(do(a,s)) = f \equiv \\ &\exists t, l.\ t = start(s) \wedge val(battLevel(s), t) = l \wedge \\ &\quad [\exists x', y'. a = startGo(\langle x', y' \rangle) \wedge f = linear(l, -\Delta_V, t) \vee \\ &\quad\ a = endGo \wedge f = constant(l) \vee \\ &\forall x', y'. a \neq startGo(x', y') \wedge a \neq endGo \wedge f = battLevel(s)]]. \end{aligned}$$

To ensure that this axiom and the above successor state axiom for *robotLoc* is well defined, we have to make sure that the value of *battLevel* in the initial situation is a 1-dimensional *t-function*, and the value of *robotLoc* a 2-dimensional *t-function*.

Summarizing, to model continuous change and time in the situation calculus, we have added four new sorts: *Real*, *Time*, *t-function* (functions of time), and *t-form* (temporal formulas). We call functional fluents whose value is a t-function *continuous fluents*. In addition, we introduced a special function *val* to evaluate *t-function*s, a new kind of primitive action *waitFor*(τ), whose argument is of sort *t-form*, together with a domain-independent precondition axiom, and a new fluent *start* (the starting time of a situation) together with a successor state axiom.

4.2 cc-Golog: a Continuous, Concurrent GOLOG Dialect

We will now introduce a variant of ConGolog, which we call cc-Golog and which is founded on our new extension of the situation calculus. The main difference between ConGolog and cc-Golog is that the semantics of the latter are adapted to the new temporal situation calculus, and in particular assures that every action is executed as soon as possible.

First, we slightly change the language by replacing the instruction $(\sigma_1 \rangle\rangle \sigma_2)$ by the construct $\mathsf{conc}(\sigma_1, \sigma_2)$. Similar to $(\sigma_1 \rangle\rangle \sigma_2)$, $\mathsf{conc}(\sigma_1, \sigma_2)$ starts the prioritized concurrent execution of σ_1 and σ_2; but unlike $(\sigma_1 \rangle\rangle \sigma_2)$, which requires both σ_1 and σ_2 to reach a final state, the parallel execution of conc stops as soon as *one* of them reaches a final state. The concurrent execution of two subplans until the first one ends is quite useful in specifying robust robot control programs. For example, it is possible to execute a low-priority plan which moves a robot from one place to another, and to concurrently execute a high-priority secondary plan that stops the robot if its gripper should ever become empty to pick up the object dropped. Another example is the outermost policy of the example program from Figure 4.2, which will never finish. Note that if the full concurrent execution of two programs σ_1 and σ_2 is desired, one can always apply conc to two programs σ_1' and σ_2' which first execute σ_1 respectively σ_2 and then wait (e.g. by an appropriate test) until the other program has finished execution.

Besides the new conc construct, cc-Golog provides another instruction that was not present in ConGolog: the $\mathsf{withCtrl}(\phi, \sigma)$ instruction. The purpose of withCtrl is to facilitate the specification of blocking policies. Intuitively, $\mathsf{withCtrl}(\phi, \sigma)$ executes the program σ as long as the condition ϕ is true, but gets blocked otherwise. The idea is that a high-priority policy can block and unblock a low-priory sub-program of the form $\mathsf{withCtrl}(\phi, \sigma)$ by modifying the truth value of ϕ; we will elaborate on the use of withCtrl in Section 4.2.2. Altogether, cc-Golog offers the following constructs:

nil	the empty program
α	primitive action or $waitFor(\tau)$
$\phi?$	wait/test action
$[\sigma_1, \sigma_2]$	sequence
$\mathsf{if}(\phi, \sigma_1, \sigma_2)$	conditional
$\mathsf{while}(\phi, \sigma)$	loop
$\mathsf{conc}(\sigma_1, \sigma_2)$	prioritized execution until any σ_i ends
$\mathsf{withCtrl}(\phi, \sigma)$	guarded execution
$\{\mathsf{proc}(P_1(\vec{v_1}), \beta_1); ...; \mathsf{proc}(P_n(\vec{v_n}), \beta_n); \sigma\}$	procedures

Note that in cc-Golog, all nondeterministic instruction found in ConGolog are missing. The reason why we had to remove them is that nondeterministic instructions are problematic with respect to cc-Golog's new semantics, as we will see in Section 4.2.4.[3]

4.2.1 A New Semantics for Concurrent Execution

To start with, the semantics remains exactly the same for all the non-concurrent constructs inherited from deterministic GOLOG, in particular for procedure definition. Note that this is also true for the new $waitFor(\tau)$ construct, which is treated like any other primitive action. When considering the transition of concurrent programs, however, care must be taken in order to avoid conflicts with the assumption that actions should happen as soon as possible, which underlies our new version of the situation calculus. To see why, let us consider the following example:

$$\mathsf{conc}([waitFor(battLevel \leq 46), chargeBatteries, \text{FALSE?}], \\ [startGo(6213), waitFor(inRoom(6213)), endGo, giveCoffee]).$$

Here, *inRoom(r)* is a macro expression that is true if and only if the value of *robotLoc* corresponds to a position inside Room r. The idea is to deliver coffee and, with higher priority, watch for a low battery level, at which point the batteries are charged. The test FALSE? in the high-priority branch is executed in order to ensure that the high-priority program will never end, and hence that the execution of the overall program will continue until the low-priority branch has completed execution. Note that a test FALSE? is neither final nor is there any possible transition to a successor configuration, and hence the execution is "blocked" forever.

In the discussion of a similar scenario written in RPL, we already pointed out that the *waitFor*-action should not block the mail delivery even though it belongs to the high priority policy. However, the obvious adaptation of ConGolog's *Trans*-definition of $(\sigma_1 \rangle\rangle \sigma_2)$ from Section 3.2 to the case of $\mathsf{conc}(\sigma_1, \sigma_2)$ (roughly, replace $(\sigma_1 \rangle\rangle \sigma_2)$ by $\mathsf{conc}(\sigma_1, \sigma_2)$ and add $\neg Final(\sigma_1, s)$ and $\neg Final(\sigma_2, s)$ as additional conjuncts in the definition's right-hand side) would yield the following trace:

$$do([waitFor(battLevel \leq 46), chargeBatteries, \\ startGo(6213), waitFor(inRoom(6213)), giveCoffee], S_0).$$

This result, where the robot first waits until its batteries are low, is clearly unacceptable. It is caused by ConGolog's *Trans* definition of $(\sigma_1 \rangle\rangle \sigma_2)$, which simply prefers the action with the higher priority. The fact that $startGo(6213)$ could immediately be executed while the

[3] Although un-prioritized concurrency is not problematic [GL00a], it is not considered here since it would also become problematic in probabilistic domains (see Section 6.4).

condition $battLevel \leq 46$ will only become true later on is not taken into account. What is required for a reasonable execution trace is that the preference is based on the *execution time* of actions, a concept which is not present in ConGolog.

As in Section 3.2.2, let us first consider a first-order definition of *Trans*, ignoring procedures. The following definition shows that it is not hard to require that actions which can be executed earlier are always preferred, restoring the original idea that actions should happen as early as possible:

$$\begin{array}{l} Trans(\mathsf{conc}(\sigma_1,\sigma_2),s,\delta,s') \equiv \\ \quad \neg Final(\sigma_1,s) \wedge \neg Final(\sigma_2,s) \wedge [\\ \qquad \exists \delta_1.\, Trans(\sigma_1,s,\delta_1,s') \wedge \delta = \mathsf{conc}(\delta_1,\sigma_2) \wedge \\ \qquad [\forall \delta_2, s_2.\, Trans(\sigma_2,s,\delta_2,s_2) \supset start(s') \leq start(s_2)] \vee \\ \qquad \exists \delta_2.\, Trans(\sigma_2,s,\delta_2,s') \wedge \delta = \mathsf{conc}(\sigma_1,\delta_2) \wedge \\ \qquad [\forall \delta_1, s_1.\, Trans(\sigma_1,s,\delta_1,s_1) \supset start(s') < start(s_1)]]. \end{array}$$

Interestingly, the two disjuncts of this axiom are almost identical. The only difference is that $\leq$ is replaced by $<$ in the last line. This ensures that σ_1 takes precedence if both σ_i are about to execute an action at the same time. We remark that this axiom stipulates that temporal precedence supersedes conc priority.

Finally, it is straightforward to give *Final* its intended meaning, that is, conc ends if one of the two programs ends:

$$Final(\mathsf{conc}(\sigma_1,\sigma_2),s) \equiv Final(\sigma_1,s) \vee Final(\sigma_2,s).$$

We remark that for *non-temporal programs*, i.e. for programs that do not involve *waitFor* actions, the new instruction $\mathsf{conc}(\sigma_1,\sigma_2)$ only differs from ConGolog's $(\sigma_1 \rangle\rangle \sigma_2)$ in that $\mathsf{conc}(\sigma_1,\sigma_2)$ becomes final as soon as any of the two programs σ_1 and σ_2 becomes final, while $(\sigma_1 \rangle\rangle \sigma_2)$ only becomes final when both σ_1 and σ_2 are final. In particular, we have the following proposition:

Proposition 4: *Let s be a legal action sequence, i.e. a situation s such that $S_0 \prec s$, and let σ_1 and σ_2 be two cc-Golog programs that do not include waitFor actions. Let Γ be the (first-order) axioms for ConGolog from Section 3.2 together with the above axiom for* conc *plus the axioms needed for the encoding of programs as first-order terms. Then:*

$$\begin{array}{l} \Gamma \models Trans(\mathsf{conc}(\sigma_1,\sigma_2),s,\delta,s') \equiv \\ \qquad Trans((\sigma_1 \rangle\rangle \sigma_2),s,\delta,s') \wedge \neg Final(\sigma_1,s) \wedge \neg Final(\sigma_2,s). \end{array}$$

Proof: If neither σ_1 nor σ_2 involve *waitFor* instructions, then by induction on the structure of σ_1:

$$Trans(\sigma_1,s,\delta_1,s') \supset s' = s \vee \exists a.[s' = do(a,s) \wedge Poss(a,s) \wedge \forall \tau. a \neq waitFor(\tau)].$$

Similarly for $Trans(\sigma_2,s,\delta_2,s'')$. By the definition of *start*, we then get

$$[Trans(\sigma_1,s,\delta_1,s') \wedge Trans(\sigma_2,s,\delta_2,s'')] \supset [start(s') = start(s) = start(s'')].$$

Thus,

$$[Trans(\sigma_1,s,\delta_1,s') \wedge [\forall \delta_2, s_2.\, Trans(\sigma_2,s,\delta_2,s_2) \supset start(s') \leq start(s_2)]]$$

is equivalent to

$$Trans(\sigma_1, s, \delta_1, s'),$$

and

$$Trans(\sigma_2, s, \delta_2, s') \wedge [\forall \delta_1, s_1. Trans(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1)]$$

is equivalent to

$$Trans(\sigma_2, s, \delta_2, s') \wedge \forall \delta_1, s_1. \neg Trans(\sigma_1, s, \delta_1, s_1).$$

□

4.2.2 Blocking Policies

As described above, in cc-Golog the preference of concurrent actions is based on their execution time, motivated by the fact that in our introductory delivery plan an action like *waitFor*(*battLevel* $\leq$ 46) should not block the concurrently running lower-priority delivery plan. Let us now turn to the case where during the execution of the introductory plan the battery level drops below 46 V. Once the routine for charging the batteries starts, it should not be interrupted, that is, it should run in *blocking mode*, unlike before.

To get a sense of the issues that arise if the robot is to execute some high-priority plans in blocking mode, let us consider the following cc-Golog procedure *chargeBatteries* which represents a possible realization of the routine for charging the batteries. *plugIn* and *plugOut* indicate that the robot connects respectively disconnects its batteries with the power supply, and the term *atPos(DockingStation)* is a macro expression that is true if and only if the value of *robotLoc* corresponds to a position near the docking station.

proc(*chargeBatteries*,
[*startGo*(*dockingStation*), *waitFor*(*atPos*(*DockingStation*)),
plugIn, *waitFor*(*battLevel* $\geq$ 49), *plugOut*])

Throughout its execution, *chargeBatteries* should not be interrupted, that is, concurrently running programs like the mail delivery routine should remain blocked. In particular, the mail delivery routine should remain blocked while *chargeBatteries* executes *waitFor*-actions it contains such as waiting for arrival at the docking station. Note that according to cc-Golog's semantics this would not be guaranteed if we would simply execute *chargeBatteries* and the mail delivery routine concurrently using conc.

To allow a natural specification of policies running in blocking mode, we have added the instruction withCtrl(ϕ, σ) which executes the program σ as long as the condition ϕ is true, but gets blocked otherwise. Using withCtrl, the effect of a policy in blocking mode can be obtained by having the truth value of ϕ be controlled by the policy and using the withCtrl(ϕ, σ)-construct in the low priority program. For example, assuming a fluent *wheels* which is initially TRUE, set FALSE by the primitive action *grabWheels* and reset by *releaseWheels*, the execution of *chargeBatteries* in blocking mode can be assured by the following plan, where *deliverMail* implements the actual mail delivery:

conc([*waitFor*(*battLevel* $\leq$ 46), *grabWheels*, *chargeBatteries*, *releaseWheels*],
withCtrl(*wheels*, *deliverMail*)).

The formal semantics of withCtrl is defined by the following axioms:

$$Trans(\mathsf{withCtrl}(\phi,\sigma),s,\delta,s') \equiv \phi[s] \wedge \exists\gamma.\, Trans(\sigma,s,\gamma,s') \wedge \delta = \mathsf{withCtrl}(\phi,\gamma);$$

$$Final(\mathsf{withCtrl}(\phi,\sigma),s) \equiv \phi[s] \wedge Final(\sigma,s).$$

To further understand the definition of withCtrl, let us first consider the case where ϕ is false. Here, $\mathsf{withCtrl}(\phi,\sigma)$ is neither *Final* nor can it ever lead to a transition. Thus, $\mathsf{withCtrl}(\phi,\sigma)$ is blocked (and not final) as long as ϕ is false. If ϕ is true, however, $\mathsf{withCtrl}(\phi,\sigma)$ behaves exactly like σ.

For convenience, in the remainder of this thesis we will also use the following macros in cc-Golog programs.[4]

$\mathsf{forever}(\sigma)$	infinite loop
$\mathsf{whenever}(\tau,\sigma)$	interrupt triggered by continuous functions
$\mathsf{withPol}(\sigma_1,\sigma_2)$	prioritized execution until σ_2 ends

Formally, these macros are defined as follows:

$$\begin{array}{rcl} \mathsf{forever}(\sigma) & \doteq & \mathsf{while}(\textsc{True},\sigma) \\ \mathsf{whenever}(\tau,\sigma) & \doteq & \mathsf{forever}([\mathit{waitFor}(\tau),\sigma]) \\ \mathsf{withPol}(\sigma_1,\sigma_2) & \doteq & \mathsf{conc}([\sigma_1,\textsc{False}?],\sigma_2). \end{array}$$

The macro $\mathsf{forever}(\sigma)$ executes a program σ over and over again. $\mathsf{whenever}(\tau,\sigma)$ is used to specify that whenever the *continuous* condition τ becomes true, the program σ is to be executed.[5] withPol is inspired by RPL's with-policy construct (cf. page 35), which has been found very useful in specifying complex concurrent behavior. Like $\mathsf{conc}(\sigma_1,\sigma_2)$, $\mathsf{withPol}(\sigma_1,\sigma_2)$ executes σ_1 and σ_2 concurrently; unlike conc, it ends if and only if σ_2 becomes final. The test False? in the definition of withPol is used to ensure that the high-priority branch will never end, and hence that the execution will continue until the low-priority branch completes execution. As in RPL, in a program $\mathsf{withPol}(\sigma_1,\sigma_2)$ we call σ_1 the policy of σ_2.

4.2.3 Extending the Semantics to Procedures

Let us now extend the semantics of cc-Golog to procedures. This can be done in complete analogy to the case of ConGolog described in Section 3.2.3. That is, we define *Trans* by the following second-order axiom:

$$Trans(\sigma,s,\delta,s') \equiv \forall T.[\ldots \supset T(\sigma,s,\delta,s')],$$

where the ellipsis stands for the universal closure of the conjunction of the following assertions:

[4] We remark that [GL00a] proposes an approach to the specification of blocking policies *without* procedures based on the use of a *macro withCtrl*(ϕ,σ) (this macro stands for σ with every primitive action or test α replaced by $\mathsf{if}(\phi,\alpha,\textsc{False}?)$). However, we do not pursue that approach here because it is not clear how it can be extended to deal with (recursive) procedures.

[5] For those familiar with [dGLL97, dGLL00], we remark that a similar idea underlies their interrupt macro $< \phi \rightarrow \sigma >$. However, while $< \phi \rightarrow \sigma >$ executes σ whenever a non-temporal condition ϕ becomes true, $\mathsf{whenever}(\tau,\sigma)$ executes σ whenever the *continuous* condition τ becomes true.

$$T(\mathsf{nil}, s, \delta, s') \equiv \textsc{False}$$
$$T(\alpha, s, \delta, s') \equiv Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s)$$
$$T(\phi?, s, \delta, s') \equiv \phi[s] \wedge \delta = nil \wedge s' = s$$
$$T([\sigma_1, \sigma_2], s, \delta, s') \equiv \exists\gamma.T(\sigma_1, s, \gamma, s') \wedge \delta = [\gamma, \sigma_2] \vee Final(\sigma_1, s) \wedge T(\sigma_2, s, \delta, s')$$
$$T(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') \equiv \phi[s] \wedge T(\sigma_1, s, \delta, s') \vee \neg\phi[s] \wedge T(\sigma_2, s, \delta, s')$$
$$T(\mathsf{while}(\phi, \sigma), s, \delta, s') \equiv \phi[s] \wedge \exists\gamma.T(\sigma, s, \gamma, s') \wedge \delta = [\gamma, \mathsf{while}(\phi, \sigma)]$$

$$\begin{aligned}
&T(\mathsf{conc}(\sigma_1, \sigma_2), s, \delta, s') \equiv \\
&\quad \neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge [\\
&\qquad \exists\delta_1.T(\sigma_1, s, \delta_1, s') \wedge \delta = \mathsf{conc}(\delta_1, \sigma_2) \wedge \\
&\qquad [\forall\delta_2, s_2.T(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)] \vee \\
&\qquad \exists\delta_2.T(\sigma_2, s, \delta_2, s') \wedge \delta = \mathsf{conc}(\sigma_1, \delta_2) \wedge \\
&\qquad [\forall\delta_1, s_1.T(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1)]]
\end{aligned}$$

$$T(\mathsf{withCtrl}(\phi, \sigma), s, \delta, s') \equiv \phi[s] \wedge \exists\gamma.T(\sigma, s, \gamma, s') \wedge \delta = \mathsf{withCtrl}(\phi, \gamma)$$

$$T(\{\mathcal{E}; \sigma\}, s, \delta, s') \equiv T(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s')$$
$$T([\mathcal{E} : P(\vec{t})], s, \delta, s') \equiv T(\{\mathcal{E}; \beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\}, s, \delta, s').$$

The only difference to the second-order definition of *Trans* in ConGolog is that the assertion for $(\sigma_1 \rangle\rangle \sigma_2)$ is replaced by the assertion for $\mathsf{conc}(\sigma_1, \sigma_2)$, that all assertions regarding nondeterministic instructions have been removed and that we have added a new assertion for $\mathsf{withCtrl}(\phi, \sigma)$.

Similarly, *Final* is defined as follows:

$$Final(\sigma, s) \equiv \forall F.[... \supset F(\sigma, s)],$$

where the ellipsis stands for the universal closure of the conjunction of the following assertions:

$$F(\alpha, s) \equiv \textsc{False} \text{ , where } \alpha \text{ is a primitive action}$$
$$F(\mathsf{nil}, s) \equiv \textsc{True}, \text{ where nil is the empty program}$$
$$F(\phi?, s) \equiv \textsc{False}$$
$$F([\sigma_1, \sigma_2], s) \equiv F(\sigma_1, s) \wedge F(\sigma_2, s)$$
$$F(\mathsf{if}(\phi, \sigma_1, \sigma_2), s) \equiv \phi[s] \wedge F(\sigma_1, s) \vee \neg\phi[s] \wedge F(\sigma_2, s)$$
$$F(\mathsf{while}(\phi, \sigma), s) \equiv \neg\phi[s] \vee F(\sigma, s)$$

$$F(\mathsf{conc}(\sigma_1, \sigma_2, s)) \equiv F(\sigma_1, s) \vee F(\sigma_2, s)$$
$$F(\mathsf{withCtrl}(\phi, \sigma), s) \equiv \phi[s] \wedge F(\sigma, s)$$

$$F(\{\mathcal{E}; \sigma\}, s) \equiv F(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s);$$
$$F([\mathcal{E} : P(\vec{t})], s) \equiv F(\{\mathcal{E}; \beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\}, s).$$

As in the case of ConGolog, it is possible to show that with respect to programs without procedures, the first-order and the second-order definitions of *Trans* and *Final* are equivalent.

Proposition 5: *With respect to cc-Golog programs without procedures, the first-order and the second-order definition of Trans and Final are equivalent.*

Proof: We prove this in complete analogy to the proof in [dGLL00] regarding ConGolog (Theorem 4). First, observe that the second-order definition of *Trans* and *Final* can be put in the following form:

$$\begin{aligned}
&\mathit{Trans}(\sigma, s, \delta, s') \equiv \\
&\quad \forall T.[\forall \sigma_1, s_1, \delta_2, s_2.\Phi_{\text{Trans}}(T, \sigma_1, s_1, \delta_2, s_2) \equiv T(\sigma_1, s_1, \delta_2, s_2)] \\
&\qquad \supset T(\sigma, s, \delta, s')
\end{aligned}$$

$$\begin{aligned}
&\mathit{Final}(\sigma, s) \equiv \\
&\quad \forall F.[\forall \sigma_1, s_1.\Phi_{\text{Final}}(F, \sigma_1, s_1) \equiv F(\sigma_1, s_1)] \supset F(\sigma, s)
\end{aligned}$$

where Φ_{Trans} and Φ_{Final} are obtained by rewriting each of the assertions in the definition of *Trans* and *Final* so that only variables appear in the left-hand part of the equations and then getting the disjunction of all right-hand sides, which are mutually exclusive since each of them deals with programs of a specific form.Then the following sentences are consequences of the second-order definitions of *Trans* and *Final*, respectively:

$$\begin{aligned}
&\forall P.[\forall \sigma_1, s_1, \sigma_2, s_2.\Phi_{\text{Trans}}(P, \sigma_1, s_1, \sigma_2, s_2) \equiv P(\sigma_1, s_1, \sigma_2, s_2)] \\
&\quad \supset \forall \sigma, s, \delta, s'.\mathit{Trans}(\sigma, s, \delta, s') \supset P(\sigma, s, \delta, s')
\end{aligned}$$

$$\begin{aligned}
&\forall P.[\forall \sigma_1, s_1.\Phi_{\text{Final}}(P, \sigma_1, s_1) \equiv P(\sigma_1, s_1)] \\
&\quad \supset \forall \sigma, s.\mathit{Final}(\sigma, s) \supset P(\sigma, s).
\end{aligned}$$

The proof is straightforward and can be found in [dGLL00]. The above sentences tells us that to prove that a property P holds for instances of *Trans* and *Final*, it suffices to prove that the property is closed under the assertions in the definition of *Trans* and *Final*, that is it suffices to show:

$$\Phi_{\text{Trans}}(P, \sigma_1, s_1, \sigma_2, s_2) \equiv P(\sigma_1, s_1, \sigma_2, s_2)$$

$$\Phi_{\text{Final}}(P, \sigma_1, s_1) \equiv P(\sigma_1, s_1).$$

Let us now denote *Trans* defined by the second-order sentence as *TransSOL* and the earlier first-order definition of *Trans* as *TransFOL*. Since procedures are not considered, we can drop, without loss of generality, the assertions for contextualized procedures and procedures with an environment in the definition of *TransSOL*. Then:

- $\mathit{TransSOL}(\sigma, s, \delta, s') \supset \mathit{TransFOL}(\sigma, s, \delta, s')$ is proven simply by noting that *TransFOL* is closed under the assertions in the definition of *TransSOL*.

- $\mathit{TransFOL}(\sigma, s, \delta, s') \supset \mathit{TransSOL}(\sigma, s, \delta, s')$ is proven by induction on the structure of σ considering as base cases nil, a, and $\phi?$, and then applying the induction argument.

Similarly for *Final*. □

This then ends the discussion of the semantics of *Final* and *Trans* in cc-Golog. *Trans** and *Do* are defined the same way as in ConGolog. As a trivial result of the fact that regarding sequential (non-concurrent) deterministic instructions the definitions for cc-Golog reflect the definitions for ConGolog, we have the following:

Corollary 6: *For sequential deterministic programs, the semantics of cc-Golog corresponds to that of ConGolog.*

4.2.4 Discussion: cc-Golog and Nondeterminism

Unlike ConGolog, cc-Golog does not provide nondeterministic instructions. We will now elaborate on the reason why we had to omit them in cc-Golog. As indicated earlier, this is because the use of nondeterminism together with cc-Golog's new semantics for concurrent execution would yield counterintuitive results. As an example, let us consider the following program: a robot is to serve a cup of coffee, and has to inform its supervisor about it either just before or after it actually gives the coffee. The possible choice is expressed using the nondeterministic branch instruction $(\sigma_1|\sigma_2)$.

$$\begin{array}{l}\mathsf{withPol}([(\mathit{waitFor}(\mathit{enterRoom})|\mathit{waitFor}(\mathit{leaveRoom})), \mathit{informSupervisor}],\\ \quad [\mathit{startGo}(\mathit{desk}), \mathit{waitFor}(\mathit{atDesk}), \mathit{give}(\mathit{coffee}),\\ \quad\ \mathit{startGo}(\mathit{home}), \mathit{waitFor}(\mathit{atHome})])\end{array}$$

Intuitively, executing this program in S_0 can lead to two different situations, reflecting the fact that the robot can inform its supervisor before or after it actually gives the coffee:

$$\begin{array}{l}\mathit{do}([\mathit{startGo}(\mathit{desk}), \mathit{waitFor}(\mathit{enterRoom}), \underline{\mathit{informSupervisor}},\\ \quad \mathit{waitFor}(\mathit{atDesk}), \mathit{give}(\mathit{coffee}), \mathit{startGo}(\mathit{home}), \mathit{waitFor}(\mathit{atHome})], S_0);\end{array}$$

and

$$\begin{array}{l}\mathit{do}([\mathit{startGo}(\mathit{desk}), \mathit{waitFor}(\mathit{atDesk}), \mathit{give}(\mathit{coffee}),\\ \quad \mathit{waitFor}(\mathit{leaveRoom}), \underline{\mathit{informSupervisor}}, \mathit{startGo}(\mathit{home}), \mathit{waitFor}(\mathit{atHome})], S_0).\end{array}$$

However, only the first situation is implied by the semantics: the second one is not allowed because it amounts to first pursuing a transition along *waitFor*(*atDesk*), which is forbidden because the nondeterministic subplan may cause an earlier transition. The situation is even worse when considering the following program, where the (1-dimensional) robot starts moving with speed 1 and then, in parallel, picks arbitrary locations, waits until it gets there, and then announces where it is:

$$\begin{array}{l}[\mathit{startGo}(1),\\ \ \mathsf{conc}([\pi x.\mathit{waitFor}(\mathit{robotLoc1d} = x), \mathit{say}(x), \mathit{set1completed}, \mathit{2completed}?],\\ \qquad [\pi y.\mathit{waitFor}(\mathit{robotLoc1d} = y), \mathit{say}(y), \mathit{set2completed}, \mathit{1completed}?]),\\ \ \mathit{endGo}].\end{array}$$

To ensure that both concurrent branches get executed, we assume fluents *1completed* and *2completed* which are initially FALSE and set TRUE by the actions *set1/2completed*, which are always possible. The testing of the fluents at the end of each concurrent branch forces that both branches need to finish.

In this example, contrary to our intuition, only one choice for x is allowed by the semantics: the starting position. This is because if x is chosen to be greater than the starting position, then there still exists a smaller y, and vice versa. Recall that just removing the guilty conditions $\forall\delta_2, s_2.\mathit{Trans}(\sigma_2, s, \delta_2, s_2) \supset \mathit{start}(s') \leq \mathit{start}(s_2)$ respectively $\forall\delta_1, s_1.\mathit{Trans}(\sigma_1, s, \delta_1, s_1) \supset \mathit{start}(s') < \mathit{start}(s_1)$ in the definition of *Trans* is no option here, because this would lead to the anomalies mentioned in Section 4.2.1.

To avoid these counterintuitive results, we have chosen to completely renounce to the use of nondeterminism in cc-Golog. We admit that this represents a serious restriction to the expressiveness of cc-Golog. On the upside, cc-Golog programs are generally much faster to execute. In these aspects, cc-Golog is more like the non-logic-based programming language RPL and its relatives. However, we remark that unlike the non-logic-based approaches, cc-Golog comes with a thorough declarative semantics. Finally, as we will see in Section 5.2.2, it is possible to specify cc-Golog plans which appeal to on-the-fly projections of sub-plans, which represents a restricted form of deliberation over different sub-plans. While the explicit use of on-the-fly projections in conditions is certainly not as elegant as the use of nondeterministic instructions (which are resolved through projection by the interpreter), it gives the user a better idea of the exact amount of reasoning (i.e. projection) involved in the execution of a plan. We will elaborate on this topic in Section 5.3.

4.3 A Robot Control Architecture

In modern robot architectures like XAVIER [SGH+97c, SGH+97a] and RHINO [BCF+00], the high-level controller does not directly operate the robot's physical sensors and effectors (cf. Section 2.3). Instead, it is connected to a basic task-execution level which provides specialized *low-level processes* like a navigation process, an object recognition process or a process for grasping objects. The job of the high-level controller is then to combine the activation and deactivation of these routines in a way to fulfill the overall goals. It turns out that cc-Golog allows a logical reconstruction of this type of architecture in a fairly natural way. The overall architecture is illustrated in Figure 4.3.

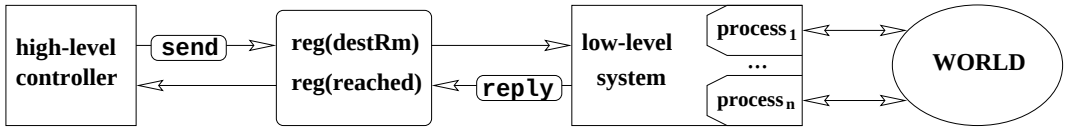


Figure 4.3: A Robot Control Architecture

4.3.1 The Communication between the High-Level Controller and the Low-Level Processes

The communication between the low-level processes and the high-level plan is achieved through the special fluent $\mathit{reg}(\mathit{id}, s)$. When no confusion arises, we will simply write $\mathit{reg}(\mathit{id})$ or "register *id*" instead of $\mathit{reg}(\mathit{id}, s)$. The high-level interpreter can affect the value of *reg* by means of the special action $\mathit{send}(\mathit{id}, \mathit{val})$, whose effect is to assign $\mathit{reg}(\mathit{id})$ the value *val*. The intuition is that in order to activate a low-level process, the high-level controller executes a *send* action. For example, the execution of $\mathit{send}(\mathit{destRoom}, 6213)$ would tell the navigation process to start moving towards Room 6213.

On the other hand, the low-level processes can provide the high-level controller with information by means of the exogenous action $reply(id, val)$.[6] For example, in order to inform the high-level controller that it has reached its destination, the low-level process would cause an exogenous $reply(arrivedAt, 6213)$ action. The following successor state axiom specifies how *reg* is affected by *send* and *reply*:

$$\begin{aligned} Poss(a, s) \supset [reg(id, do(a, s)) = val \equiv \\ a = send(id, val) \vee a = reply(id, val) \vee \\ reg(id, s) = val \wedge \neg\exists v.(a = send(id, v) \vee a = reply(id, v))]. \end{aligned}$$

send and *reply* are always executable, that is $Poss(send(id, val)) \equiv \text{TRUE}$ and similarly $Poss(reply(id, val)) \equiv \text{TRUE}$. In the remainder of this thesis, we will use AX_{arch} to refer to the successor state axioms for *reg* together with axioms stating that *reply* and *send* are always possible. Furthermore, we use the convention that initially all registers have value nil, and thus add $\forall id.reg(id, S_0) = \text{nil}$ to AX_{arch}. In particular, this means that no messages have been issued in S_0.

4.3.2 Modeling Low-Level Processes as cc-Golog Procedures

Most low-level processes in real-world applications need to be described at a level of detail involving many atomic actions interacting in complicated ways. To describe complex low-level processes like a navigation process, we model them as cc-Golog *procedures*. We thereby abandon the simplified view of the previous sections where we assumed that the high-level controller directly operates the physical devices. Given a faithful characterization of the low-level processes in terms of cc-Golog procedures, we can then *project* the effect of the activation of these processes using their corresponding cc-Golog models. We stress that these procedures are not meant to be executed, but rather represent a model of the effects of the corresponding low-level process.

As an example, let us consider the low-level navigation process. It is activated through a $send(destRoom, r)$ action, which assigns $reg(destRoom)$ the value r and tells the navigation process to travel to room r. The navigation process then starts moving the robot to its desired location, making use of intermediate goal points like those outside doorways for robustness. The process remains active until r is reached or $reg(destRoom)$ is assigned a new value. In the former case, it informs the high-level controller of the arrival by means of a $reply(arrivedAt, r)$. In the latter case, the navigation process aborts its journey and immediately restarts with the new destination.

We model this behavior by the cc-Golog procedure *navProc*. *navProc* makes use of the defined functional fluent *currentRoom*, which is defined in terms of *robotLoc* and whose value in a situation s is the name of the room the robot is in at the beginning of s. Initially, *navProc* is blocked until $reg(destRoom)$ is assigned a (non-nil) room name different from the one the robot is actually in. If $reg(destRoom)$ is assigned a new value, *navProc* calls the procedure $travelTo(reg(destRoom))$ which simulates the behavior of the low-level navigation process while travelling to a new destination (see the description below). After the execution of *travelTo*, *navProc* returns to its initial state. We remark that *navProc* will run forever. This is fine because we will never consider the mere projection of *navProc*, but instead of a

[6] As mentioned in the introduction (i.e. Chapter 1), we consider actions as exogenous if they are not under the control of the high-level controller. We remark that *reply* actions represent a kind of sensing, as we will elaborate in the next chapters.

high-level cc-Golog plan running concurrently to *navProc*; the projection will only be pursued until the completion of the high-level plan.

$$\mathsf{proc}(gotoLoc(\langle x, y\rangle), [startGo(\langle x, y\rangle), waitFor(near(\langle x, y\rangle))])$$

$$\begin{aligned}\mathsf{proc}(travelTo(dest),\mathsf{conc}([&\mathsf{if}(currentRoom \neq \text{HALLWAY},\\ &\quad [gotoLoc(exitOf(currentRoom)),\\ &\quad\ gotoLoc(entryOf(currentRoom))]),\\ &\ gotoLoc(entryOf(dest)),\\ &\ gotoLoc(exitOf(dest)),\\ &\ gotoLoc(centreOf(dest)),\\ &\ endGo, reply(arrivedAt, dest)],\\ &[(reg(destRoom) \neq dest)?, endGo]))\end{aligned}$$

$$\begin{aligned}\mathsf{proc}(navProc, \mathsf{forever}([&(reg(destRoom) \neq currentRoom \wedge reg(destRoom) \neq \mathsf{nil})?,\\ &travelTo(reg(destRoom))]))\end{aligned}$$

The procedure *travelTo* makes us of the following functions: $exitOf(r)$, $entryOf(r)$ and $centreOf(r)$. *exitOf* maps a room name r to (the coordinates $\langle x, y\rangle$ of) a location inside room r next to its exit, *entryOf* to a location outside r and *centreOf* to a location within r. *travelTo* concurrently starts travelling towards the destination room and checking whether it is aborted. The latter is realized by the second branch of conc which immediately executes *endGo* and becomes final if $reg(destRoom)$ is assigned a new value. If this happens, the second branch of conc becomes unblocked due to cc-Gologs *call-by-value* parameter passing, inherited from ConGolog (cf. Section 3.2.3). After the execution of *endGo*, the whole conc becomes final.

Using $exitOf(r)$, $entryOf(r)$ and $centreOf(r)$, the trajectory is approximated by a polyline with an edge in front and behind every door the robot has to travel through (see Figure 4.4). The procedure $gotoLoc(\langle x, y\rangle)$ causes the robot to travel to the coordinates $\langle x, y\rangle$. The term $near(\langle x, y\rangle)$ is a macro expression that is true if the value of *robotLoc* is sufficiently close to $\langle x, y\rangle$, typically within a few centimeters.

4.3.3 Projection

We will now describe how to project a cc-Golog plan, taking into account the cc-Golog model of the low-level processes. Let s be a situation, ll_{model} a model of the low-level processes, and σ a cc-Golog program. We identify a projection with the situation s' that results from the concurrent simulation of σ and ll_{model} until σ ends, starting in s:

$$Proj(s, \sigma, ll_{model}, s') \doteq Do(s, \mathsf{withPol}(ll_{model}, \sigma), s').$$

Note that although we represent the low-level processes as running with higher priority than the high-level plan, this priority ordering has no deep impact on the resulting projections. This is because we consider processes with temporal extent, where the different priorities manifest only when two processes wish to execute an action at exactly the same time; actions with different execution times are not affected.

As an example, let us consider a projection of the following simple plan Π_{simple}. We write *inHallway* as an abbreviation for an expression that is true if the value of *robotLoc* corresponds to a position in the hallway.

$$\Pi_{simple} \doteq \mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ [send(destRoom, 6205), (reg(arrivedAt) = 6205)?])$$

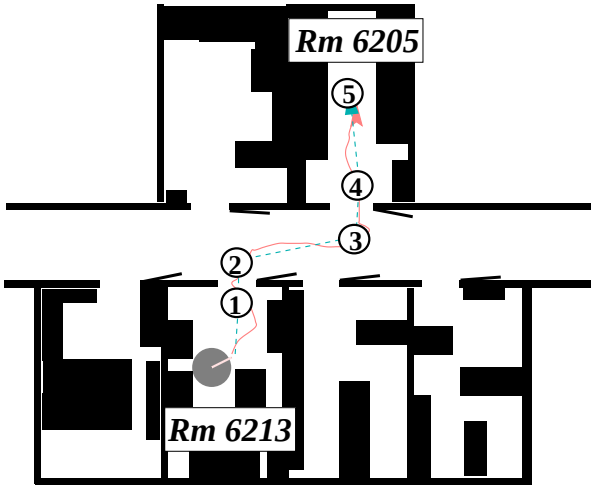


Figure 4.4: Piece-wise Linear Approximation of the Actual Trajectory

In this example, the navigation process is the only low-level process, so we can simply use *navProc* as ll_{model} (more precisely, we would use $\{\mathcal{E}, navProc\}$ where the environment $\mathcal{E}$ comprises the procedure definitions of Section 4.3.2; cf. the discussion on Page 53). If there were more relevant low-level processes, ll_{model} would amount to the concurrent execution of the individual cc-Golog models (cf. Sections 6.2.2 and 6.2.3). Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{cc}}$ from the previous chapter, axioms for *val* defining the value of the *t-function*s, the set of axioms $\mathrm{AX}_{\mathrm{arch}}$, the axioms for cc-Golog (cf. Sections 4.2.1, 4.2.2 and 4.2.3), the axioms needed for the encoding of cc-Golog programs as first-order terms (cf. Appendix A), initial state axioms stating that the robots initial position is *constant* and lies within Room 6213 and that the initial battery voltage is sufficiently high, successor state axioms specifying how the robot's location and battery level are affected by *startGo* and *endGo* (cf. Section 4.1.5), and precondition axioms stating that *endGo* is always possible and *startGo* is only possible if the destination differs from the robot's current location. Using Γ, we can project the plan Π_{simple}:

$$\begin{aligned}\Gamma \models\ & Proj(S_0, \Pi_{simple}, navProc, s) \equiv \\ & s = do([send(destRoom, 6205), startGo(l_1), \\ & \quad waitFor(near(l_1)), startGo(l_2), \\ & \quad waitFor(inHallway), say(\text{“in Hallway”}) \\ & \quad waitFor(near(l_2)), startGo(l_3), \\ & \quad waitFor(near(l_3)), startGo(l_4), \\ & \quad waitFor(near(l_4)), startGo(l_5), \\ & \quad waitFor(near(l_5)), endGo, \\ & \quad reply(arrivedAt, 6205)], S_0).\end{aligned}$$

Before we present a proof sketch, we remark that the projected execution trace includes a $startGo(l_i)$ and $waitFor(near(l_i))$ action for every node on the approximating trajectory, where the l_i stand for coordinates corresponding to the nodes i in Figure 4.4. Additionally,

it includes a *say*("in Hallway") which is executed immediately after the hallway has been reached, which is assumed to happen just after leaving $near(l_1)$. The execution trace ends with $reply(arrivedAt, 6205)$, which completes the navigation task.

Proof: (Sketch) Just as in the ConGolog example of Section 3.2.2, the proof is straightforward but quite laborious. Essentially, it amounts to proving that there is a sequence of transitions from $\langle \mathsf{withPol}(ll_{model}, \Pi_{simple}), S_0 \rangle$ to a final configuration with the above projection trace as situation component. We will only consider some of the transitions along the sequence. In particular, we will focus on the interaction of the prioritized model *navProc* and the low-priority plan Π_{simple}, and on the impact of cc-Golog's new semantics for concurrent execution.

As mentioned above, strictly speaking we do not use *navProc* as the model of the low-level navigation process but $\{\mathcal{E}, navProc\}$, where the environment $\mathcal{E}$ comprises the procedure definitions of Section 4.3.2. However, in the following we simply leave out the environment. Initially, *navProc* is blocked because $reg(destRoom) = \mathsf{nil}$. On the other hand, Π_{simple} is ready to execute $send(destRoom, 6205)$. So we get:

$$\begin{array}{l} Trans(\mathsf{withPol}(navProc, \Pi_{simple}), S_0, \delta, s) \equiv \\ \quad s = do(send(destRoom, 6205), S_0) \wedge \\ \quad \delta = \mathsf{withPol}(navProc, \\ \qquad \mathsf{withPol}([waitFor(inHallway), say(\text{"in Hallway"})], \\ \qquad\quad [\mathsf{nil}, (reg(arrivedAt) = 6205)?])). \end{array}$$

That is, there is a transition to a configuration with situation component

- $S_1 \doteq do(send(destRoom, 6205), S_0)$

and program component

- $\mathsf{withPol}(navProc,$
  $\mathsf{withPol}([waitFor(inHallway), say(\text{"in Hallway"})],$
  $[\mathsf{nil}, (reg(arrivedAt) = 6205)?]))$.

This is the only successor configuration that can be directly reached via *Trans*. In the new configuration, the high-priority process *navProc* gets unblocked, because

$$currentRoom(S_1) = 6213 \wedge reg(destRoom, S_1) = 6205.$$

Thus, *navProc* can cause an immediate transition involving the evaluation of its initial test $(reg(destRoom) \neq currentRoom \wedge reg(destRoom) \neq \mathsf{nil})?$. As *navProc* is the high-priority branch and the transition is immediate, the definition of *Trans* implies that there is a transition from the above configuration to a new configuration $\langle s', \delta \rangle$ if and only if:

- $s' = S_1$, and
- $\delta = \mathsf{withPol}([[\mathsf{nil}, travelTo(reg(destRoom))], navProc],$
  $\mathsf{withPol}([waitFor(inHallway), say(\text{"in Hallway"})],$
  $[\mathsf{nil}, (reg(arrivedAt) = 6205)?]))$.

Next, from the definition of *Trans* and the fact that $startGo(l_1)$ is possible, we can conclude that there is a transition from the above configuration to a new configuration $\langle s', \delta \rangle$ if and only if:

- $s = S_2 \doteq do(startGo(l_1), S_1)$, and
- $\delta = \mathsf{withPol}([\mathsf{conc}([[[\mathsf{nil}, waitFor(near(l_1))],$
  $gotoLoc(entryOf(currentRoom))],$
  $gotoLoc(entryOf(6205)),$
  $gotoLoc(exitOf(6205)),$
  $gotoLoc(centreOf(6205)),$
  $endGo, reply(arrivedAt, 6205)]$
  $[(reg(destRoom) \neq 6205)?, endGo]),$
  $navProc],$
  $\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})],$
  $[\mathsf{nil}, (reg(arrivedAt) = 6205)?])).$

Note that the outermost policy corresponds to the model of *navProc*, while the last two lines represent what remains of Π_{simple}. In this configuration, both the outermost high-priority policy representing the navigation process and the plan's policy can cause a transition: the former involves the execution of $waitFor(near(l_1))$, and the latter of $waitFor(inHallway)$. The low-priority branch of the plan can not cause a transition because $reg(arrivedAt, S_2) \neq 6205$. As Figure 4.4 illustrates, the least time point of the *t-form* $near(l_1)$ is earlier than that of the *t-form inHallway*: the value of $robotLoc(S_2)$ is a linear function of time whose orientation corresponds to the dotted line from the robot's initial position to node (1), and this line crosses node (1) before it crosses the hallway. Thus, according to the new semantics for concurrent execution, the high-priority process takes precedence. That is, there is a transition from the above configuration to a new configuration $\langle s', \delta \rangle$ if and only if:

- $s = S_3 \doteq do(waitFor(near(l_1)), S_2)$, and
- $\delta = \mathsf{withPol}([\mathsf{conc}([[\mathsf{nil},$
  $gotoLoc(entryOf(currentRoom))],$
  $gotoLoc(entryOf(6205)),$
  $gotoLoc(exitOf(6205)),$
  $gotoLoc(centreOf(6205)),$
  $endGo, reply(arrivedAt, 6205)]$
  $[(reg(destRoom) \neq 6205)?, endGo]),$
  $navProc],$
  $\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})],$
  $[\mathsf{nil}, (reg(arrivedAt) = 6205)?])).$

We remark that while so far the starting time was unaffected by the transitions, the execution of $waitFor(near(l_1))$ causes *start* to advance to the least time point of the *t-form* $near(l_1)$. The next transition is caused by the high-priority process which executes $startGo(l_2)$: there is a transition from the above configuration to a new configuration $\langle s', \delta \rangle$ if and only if:

- $s = S_4 \doteq do(startGo(l_2), S_3)$, and
- $\mathsf{withPol}([\mathsf{conc}([[\mathsf{nil}, waitFor(near(l_2))],$
  $gotoLoc(entryOf(6205)),$
  $gotoLoc(exitOf(6205)),$
  $gotoLoc(centreOf(6205)),$

$$\begin{aligned}
&\quad\quad endGo, reply(arrivedAt, r)]\\
&\quad\quad [(reg(destRoom) \neq 6205)?, endGo]),\\
&\quad\quad navProc],\\
&\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})],\\
&\quad\quad [\mathsf{nil}, (reg(arrivedAt) = 6205)?])).
\end{aligned}$$

Once again, both the outermost high-priority policy representing the navigation process and Π_{simple}'s policy can cause a transition in this configuration. The former involves the execution of *waitFor*(*near*(l_2)), and the latter of *waitFor*(*inHallway*). This time, as illustrated by Figure 4.4, the least time point of the *t-form inHallway* is earlier than that of the *t-form near*(l_2). That is, the lower-priority process can cause a transition before the high-priority process. According to the new semantics for concurrent execution, the earlier transition is pursued, that is the lower-priority plan may execute before the high-priority model of the low-level navigation process. Thus, there is a transition from the above configuration to a new configuration $\langle s', \delta \rangle$ if and only if:

- $s = S_5 = do(waitFor(inHallway), S_4)$, and
- $$\begin{aligned}
&\mathsf{withPol}([\mathsf{conc}([[\mathsf{nil}, waitFor(near(l_2))],\\
&\quad\quad gotoLoc(entryOf(6205)),\\
&\quad\quad gotoLoc(exitOf(6205)),\\
&\quad\quad gotoLoc(centreOf(6205)),\\
&\quad\quad endGo, reply(arrivedAt, r)]\\
&\quad\quad [(reg(destRoom) \neq 6205)?, endGo]),\\
&\quad\quad navProc],\\
&\mathsf{withPol}([\mathsf{nil}, say(\text{“in Hallway”})],\\
&\quad\quad [\mathsf{nil}, (reg(arrivedAt) = 6205)?])).
\end{aligned}$$

Here, the lower-priority plan executes another action, namely *say*(“in Hallway”) which can immediately be executed. Thereafter, the high-priority branch corresponding to the navigation process resumes execution. We do not consider these transitions in detail. Finally, we come to a final configuration whose situation component corresponds to the overall execution trace, which together with the definitions of *Trans**, *Do* and *Proj* finishes the proof. □

4.3.4 The Example Revisited

This leads us, finally, to the specification of our initial example in cc-Golog. Again, we assume a fluent *wheels*, which is initially TRUE, set FALSE by *grabWheels*, and reset by the action *releaseWheels*.

$$\begin{aligned}
\Pi_{intro} \doteq\ &\mathsf{withPol}(\mathsf{whenever}(battLevel \leq 46,\\
&\quad\quad [grabWheels, chargeBatteries, releaseWheels]),\\
&\quad \mathsf{withPol}(\mathsf{whenever}(nearDoor(\text{r6213}),\\
&\quad\quad [say(\text{“hello”}), waitFor(\neg nearDoor(\text{r6213}))]),\\
&\quad\quad \mathsf{withCtrl}(wheels, deliverMail)))
\end{aligned}$$

In this program, *nearDoor*(r6213) is a macro which is true if the robot's position is near Door 6213. The outermost policy is waiting until the battery level drops to 46. At this point, a *grabWheels* is immediately executed, which blocks the execution of the program *deliverMail*.

It is only after the complete execution of *chargeBatteries* that *wheels* gets released so that *deliverMail* may resume execution (if, while driving to the battery docking station, the robot passes by Room 6213, it would still say "hello"). Note that the cc-Golog program is in a form very close to the original RPL-program we started out with. Hence we feel that cc-Golog is a step in the right direction towards modeling more realistic domains which so far could only be dealt with in non-logic-based approaches. Moreover, with their rigorous logical foundation, it is now possible to make provable predictions about how the world evolves when executing cc-Golog programs.

In order to complete the example, we only need to specify the procedures *chargeBatteries* and *deliverMail*. In this example, new subtleties arise due to the fact that the high-priority policy might interrupt a low-level process during execution. In particular, *deliverMail* might be interrupted by *chargeBatteries* due to low voltage level while travelling to a room r. If this happens, *chargeBatteries* will block the further execution of *deliverMail* and will immediately tell the navigation process to abort its actual task and instead travel to the docking station. It is only after completition of *chargeBatteries* that *deliverMail* will become unblocked. By that time, however, the low-level navigation process will have completed execution – the robot is already near the docking station. That is, the navigation process will not cause the robot to move towards r. So if *deliverMail* would simply wait for $reg(arrivedAt) = r$ like Π_{simple} (cf. page 73), it would get blocked forever.

In order to avoid this deadlock, *deliverMail* has to realize that the navigation process is idle, and has to reactivate it. The following cc-Golog procedure represents a possible solution to an interruptible high-level navigation routine leading the robot to room r. It makes use of the fact that the condition $reg(arrivedAt) = reg(destRoom)$ can be used to determine whether the low-level process has finished execution. If after completion of the navigation process *currentRoom* is different from r, then the navigation process is activated once again. Thus, $gotoRoom(r)$ only ends if the robot has actually reached room r.

$$\begin{array}{l}\mathsf{proc}(gotoRoom(r),\\ \quad \mathsf{while}(currentRoom \neq r,\\ \qquad [send(destRoom, r), reg(arrivedAt) = reg(destRoom)?]))\end{array}$$

The following code sketches a possible realization of *deliverMail*, which makes use of *gotoRoom* to assure that the delivery is interruptible. It is defined in terms of the functional fluent *letterQueue*, which intuitively represents a queue whose elements are the letters that are to be delivered. We assume a predicate $empty(letterQueue)$, which is true if and only if no letter is queued in *letterQueue*, and two functions $top(letterQueue)$ and $destination(l)$, whose values correspond to the first element of *letterQueue* respectively to the name of the destination of letter l. Furthermore, we assume appropriate successor state axioms which ensure that the effect of $push(l, letterQueue)$ and $pop(letterQueue)$ is to enqueue a letter, respectively to remove the first element of *letterQueue*.

$$\begin{array}{l}\mathsf{proc}(deliverMail,\\ \quad \mathsf{while}(\neg empty(letterQueue),\\ \qquad [gotoRoom(destination(top(letterQueue))),\\ \qquad say(\text{“hello, i got mail for you”}),\\ \qquad \text{... /* wait for the user to get her letter */}\\ \qquad pop(letterQueue)]))\end{array}$$

Finally, the following procedure replaces the earlier definition on page 65, which did not account for the fact that the high-level plans do not directly operate the robot's physical sensors and effectors.

$$\mathsf{proc}(\mathit{chargeBatteries},\\ [\mathit{gotoRoom}(\mathit{dockingStation}), \mathit{plugIn}, \mathit{waitFor}(\mathit{battLevel} \geq 49), \mathit{plugOut}])$$

Figure 4.5 illustrates a possible projection trace of the example plan Π_{intro}, where the robot is to deliver two letters to Room 6205 respectively to Room 6214. The robot first delivers a letter to Room 6205 ("1"), and then heads towards Room 6214. During this route, the battery level drops below 46 V, and the mail delivery is interrupted by *chargeBatteries* ("2"). After recharging its batteries ("3"), *deliverMail* is reactivated. The routine *gotoRoom* realizes that the journey towards Room 6214 has been aborted, and issues another $\mathit{send}(\mathit{destRoom}, 6214)$, which causes the robot to move to Room 6214 and complete its journey ("4").

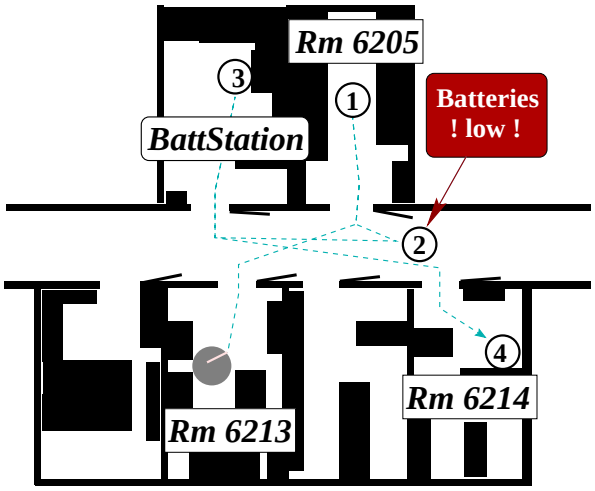


Figure 4.5: Projection of the mail delivery plan

4.4 Discussion

In this chapter, we have presented an extension of the situation calculus which includes a model of continuous change and a novel approach to modeling the passage of time. The presentation in the sections 4.1.1, 4.1.2 and 4.1.3 essentially follows Pinto [Pin97], in a somewhat simplified form. We then considered cc-Golog, a deterministic variant of ConGolog which is based on the extended situation calculus. A key feature of the new language is the ability to have part of a program wait for an event like the battery voltage dropping dangerously low while other parts of the program run in parallel. For that purpose, we have introduced the special action *waitFor*, motivated by the fact that a similar instruction is present in many special robot programming languages, in particular in RPL, RAP, PRS-Lite and Colbert. Such mechanisms allow very natural formulations of robot controllers, in particular, because there is no need to state explicitly in the program when actions should occur. On the downside, cc-Golog does not provide nondeterministic instructions.

We remark that *waitFor* cannot be simulated by means of the instructions already present in ConGolog. Although it may seem suggestive to make use of tests ($\phi?$) to react to conditions like low voltage level, this does not result in the intended behavior. The problem is that continuously changing conditions are neither true nor false in a given situation, but instead become true at a certain time point. For example, imagine we would run the test *nearDoor*(6204)? concurrently to *deliverMail*. Even though *deliverMail* will cause the robot to come near this door, the *nearDoor* condition might never be true at the beginning of a situation during the execution trace.

Of course, one could make use of an explicit "clock" which causes time to advance by means of a sequence of discrete "clock tick" actions. However, one objection against the use of a clock is that it is not clear what granularity to choose – the right granularity may depend on the application as well as on the high-level plan considered. Furthermore, the resulting execution traces would be glutted with irrelevant "clock tick" actions. This is particularly undesirable because a large number of clock tick actions may significantly decrease the performance of an implementation. Note that after *every* clock tick action the interpreter would have to check whether that action unblocks *any* of the concurrently running sub-plans.

We end this section with some remarks on Reiter's proposal for a temporal version of GOLOG [Rei98], which makes use of a different temporal extension of the situation calculus [Rei96]. Roughly, the idea is that every primitive action has as an extra argument its execution time. For example, we would write *endGo*(20) to indicate that *endGo* is executed at time 20. It turns out that this explicit mention of time is problematic when it comes to formulating programs such as the one from Section 4.3.4. Consider the part about saying "hello" whenever the robot is near Room 6213. In Reiter's approach, the programmer would have to supply a temporal expression as an argument of the *say*-action. However, it is far from obvious what this expression would look like since it involves analyzing the mail delivery sub-program as well as considering the odd chance of a battery recharge. In a nutshell, while Reiter's approach forces the user to figure out when to act, we let cc-Golog do the work. — As a final aside, we remark that *waitFor*-actions allow us to easily emulate Reiter's approach within our framework by using a continuous fluent *time* with value $linear(0, 1, 0)$ and replacing every dated action $a(\vec{x}, t)$ by the sequence $[waitFor(time = t), a(\vec{x})]$.

Chapter 5

On-Line Execution of cc-Golog Plans

Using the high-level language cc-Golog introduced in the last chapter, it is possible to provide quite natural formulations of robot controllers, in particular to specify event-driven actions which react to conditions on the value of continuously changing properties like the robot's position. So far, however, we have only shown how to *project* event-driven cc-Golog plans, making use of an extended situation calculus capable of representing continuous change. Up to now, it remains unclear how to actually *execute* a cc-Golog plan.

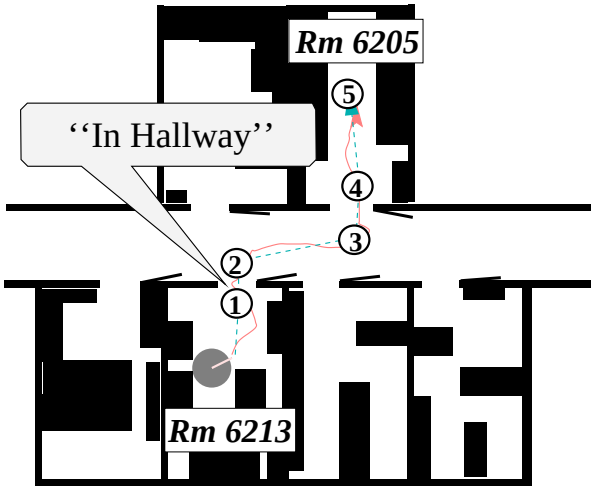


Figure 5.1: Piece-wise Linear Approximation and Actual Trajectory

To get a sense of the problem, consider a cc-Golog plan telling the robot to move from Room 6213 to Room 6205, and concurrently to react to the continuous condition *inHallway* by announcing that it has just reached the hallway (see Figure 5.1). In the previous chapter, we have shown how, based on a model of the robot control architecture and the low-level processes, it is possible to generate projections of such cc-Golog plans. In particular, the projected execution traces include *waitFor* actions for every node on the approximating trajectory. While the resulting execution traces can be understood as a way of assessing whether a program is executable *in principle*, they are *not* suitable as input to the execution mechanism of the robot for several reasons.

For one, many of the actions in the projected execution trace only serve to model the

navigation process and are not meant to be executed by the high-level controller at all. For example, the *waitFor* actions corresponding to the nodes on the approximating trajectory result from the simulation of a cc-Golog program which models the effects of the level-low navigation process. During actual execution, the controller should only issue a message to the low-level navigation process telling it the new destination (i.e. node 5 in Figure 5.1), and the navigation process would then move the robot to its desired location. For another, the time point of actions that do belong to the high-level controller like *waitFor*(*inHallway*) or *waitFor*($battLevel \leq 46$) must be interpreted differently during on-line execution. This is because during projection, the time point of a *waitFor*-condition like *inHallway* or $battLevel \leq 46$ is computed based on an idealized *model* of the world (like piece-wise linear trajectories and constant energy consumption). However, the robot has no control over the passage of time, and the actual trajectory often follows a function quite different from the idealized approximation (see the curved vs. the straight (dotted) line in Figure 5.1). During actual execution, of course, the robot should react at the *actual* time where a condition becomes true.

What is needed, it seems, are frequent sensor readings telling the robot about the current time and location, which should be used instead of the models of how time passes or how the robot moves. For example, the robot should react to the *actual* battery level by periodically reading its voltage meter. A *waitFor* action would then simply reduce to a test, where a condition like being at a certain location is matched against sensor readings reflecting the actual state of affairs. That is, we opt for an on-line style of execution (cf. Section 2.2) where we distinguish between projection-time and execution-time effects. Thereby, we take advantage from the fact that in our robot control architecture from Section 4.3 there is a clear separation of the actions of the high-level controller from those of low-level processes like the navigation process, and completely ignore the model of the low-level processes during execution.

Apart from this, while projection in cc-Golog so far is limited to a complete program starting in the initial state, one would often like to project on the fly during execution, similar to the search operator of de Giacomo and Levesque [dGL99b]. However, for reasons of limited resources, we would like to go beyond that and allow for a restricted projection of a program, which only searches up to a (temporally) limited horizon. For instance, if the robot is in the middle of a delivery but near the docking station, we want to enable it to find out whether the coming activities would allow it to operate for at least another 5 minutes and, if not, decide to charge the batteries first.

In this chapter, we show how all this can be done in cc-Golog. The chapter is organized as follows: in the next section, we discuss the changes necessary to use the same cc-Golog program both for on-line execution and projection. Thereafter, we show how on-line execution and projection can be interleaved, and introduce a time-bounded projection mechanism that can be used to appeal to on-the-fly projection in cc-Golog programs.

5.1 On-Line Execution of cc-Golog Plans

As mentioned above, during on-line execution of a cc-Golog plan it seems appropriate to make use of frequent sensor readings telling the robot about the current time and location, instead of using the models of how time passes or how the robot moves. In order to see how such sensor readings can be obtained, let us briefly look at how actual robots like RHINO [Bee99, BCF+00]

or MINERVA [TBB+99] deal with it. There, we find a tight update loop between the low-level system and the high-level controller. This update loop periodically provides the high-level controller with an update of the low-level processes' estimates of continuous properties like the batteries' voltage level or the robot's position (typically several times a second). The period of time between two subsequent updates is so small that for practical purposes the latest update can be regarded as providing not only the correct current time but also accurate values of the continuous fluents at the current time.

5.1.1 *ccUpdate* - Updating Continuous Fluents

Our solution is then to represent the updates by means of a new exogenous action *ccUpdate* and to treat *waitFor*'s simply as special tests during on-line execution. Intuitively, the effect of *ccUpdate* is to set the value of the continuous fluents to the latest estimates of the low-level processes. On the other hand, we completely ignore the cc-Golog model of the low-level processes during (on-line) execution.

The actual arguments of *ccUpdate* depend on the continuous fluents that are to be updated. In our example scenario, *ccUpdate* has four arguments: x, y, l and t, where x and y refer to the current position of the robot, l to the current voltage level and t to the current time.[1] To get a sense of the effect of *ccUpdate*, let us consider a new version of the successor state axiom for *robotLoc* of Section 4.1.5 suitable for both on-line execution and projection. We assume that *ccUpdate* never occurs during projection (in particular, *ccUpdate* may not be used in the cc-Golog models of the low-level processes). Similarly, we assume that the actions *startGo* and *endGo* only occur during projection, where they are part of the model of the low-level navigation process.

$$
\begin{aligned}
Poss(a,s) \supset [&robotLoc(do(a,s)) = f \equiv \\
&\exists t,x,y.\ t = start(s) \wedge val(robotLoc(s),t) = \langle x,y\rangle \wedge \\
&\quad [\exists x',y',v_x,v_y.a = startGo(\langle x',y'\rangle) \wedge \\
&\qquad v_x = (x'-x)/\nu \wedge v_y = (y'-y)/\nu \wedge \wedge f = linear(x,y,v_x,v_y,t) \vee \\
&\quad a = endGo \wedge f = constant(x,y) \vee \\
&\quad \exists x',y',l',t'.a = ccUpdate(x',y',l',t') \wedge f = constant(x',y') \vee \\
&\forall \vec{x}.a \neq startGo(\vec{x}) \wedge a \neq ccUpdate(\vec{x}) \wedge a \neq endGo \wedge f = robotLoc(s)]
\end{aligned}
$$

The first five lines of this axiom correspond to those of the old version of the successor state axiom for *robotLoc* from Section 4.1.5. In particular, the variables x and y refer to the actual coordinates of the robot; after $startGo(\langle x',y'\rangle)$, *robotLoc* has as value a *linear t-function* moving towards $\langle x',y'\rangle$; and after *endGo*, it is $constant(\langle x,y\rangle)$. As before, $\nu \doteq \sqrt{(x'-x)^2+(y'-y)^2}$ is a normalizing factor which ensures that the total 2-dimensional velocity is 1.

Let us now consider the new case where a is a $ccUpdate(x',y',l',t')$ action. In this case, which is handled by the sixth line of the above axiom, the value of *robotLoc* after execution of a is simply the constant function $constant(x',y')$. This reflects the idea that, during on-line execution, the linear approximation of the trajectory plays no role and that the actual values $\langle x',y'\rangle$ should be used instead, where $\langle x',y'\rangle$ is the latest estimates of the low-level navigation process. Finally, the last line of the above axiom say that if a is neither a *startGo*, an *endGo* nor a *ccUpdate* action, *robotLoc* remains unchanged.

[1] Strictly speaking, *ccUpdate* is an action schema. The actual signature of *ccUpdate* depends on the specific low-level execution system that is to be coupled with cc-Golog.

Similarly, the following successor state axiom for *battLevel* is suitable for both on-line execution and projection:

$$\begin{aligned}
&Poss(a,s) \supset [battLevel(do(a,s)) = f \equiv \\
&\quad \exists t, l.\ t = start(s) \wedge val(battLevel(s), t) = l \wedge \\
&\quad\quad [\exists x', y'. a = startGo(\langle x', y' \rangle) \wedge f = linear(l, -\Delta_V, t) \vee \\
&\quad\quad\ a = endGo \wedge f = constant(l) \vee \\
&\quad\quad\ \exists x', y', l', t'. a = ccUpdate(x', y', l', t') \wedge f = constant(l') \vee \\
&\quad \forall \vec{x}. a \neq startGo(\vec{x}) \wedge a \neq endGo \wedge a \neq ccUpdate(\vec{x}) \wedge f = battLevel(s)]].
\end{aligned}$$

5.1.2 The Passage of Time During On-Line Execution

Besides updating the value of the continuous fluents, *ccUpdate* is of prime importance because it *causes time to advance* during execution – *waitFor* can no longer take on this role because it makes use of an idealizing model. Therefore, we need to modify the successor state axiom of the fluent *start*. In order to account for the fact that time advances differently in projections and during on-line execution, we need to explicitly distinguish between the two modes of operation. For that purpose, we introduce a special fluent $online(s)$ which can only change truth values by the special actions *setOnline* and *clipOnline*, respectively:

$$Poss(a,s) \supset [online(do(a,s)) \equiv a = setOnline \vee [a \neq clipOnline \wedge online(s)]].$$

setOnline and *clipOnline* are always possible, that is $Poss(setOnline) \equiv \text{TRUE}$ and similarly $Poss(clipOnline) \equiv \text{TRUE}$. We remark that *setOnline* and *clipOnline* are special actions whose purpose is to allow both on-line execution and projection of the same cc-Golog plan. We assume that neither *setOnline* nor *clipOnline* ever occur within a cc-Golog plan. Using *online*, we can formally define how *start* changes its value both in projection and in on-line execution mode:

$$\begin{aligned}
Poss(a,s) \supset [&start(do(a,s)) = t \equiv \\
&\exists \tau. a = waitFor(\tau) \wedge \neg online(s) \wedge ltp(\tau, s, t) \vee \\
&\exists x_u, y_u, l_u. a = ccUpdate(x_u, y_u, l_u, t) \wedge online(s) \vee \\
&[\forall \tau. a \neq waitFor(\tau) \vee online(s)] \wedge [\forall \vec{x}. a \neq ccUpdate(\vec{x}) \vee \neg online(s)] \\
&\quad \wedge t = start(s)].
\end{aligned}$$

The precondition axiom for *ccUpdate* ensures that the starting time of legal action sequences is monotonically nondecreasing, just as in Section 4.1.4:

$$Poss(ccUpdate(x, y, l, t), s) \equiv t \geq start(s)).$$

5.1.3 On-Line Execution of *waitFor* Instructions

During on-line execution, a $waitFor(\tau)$ instruction is treated as a special kind of *test*: it succeeds immediately if and only if the condition τ is true *at the beginning* of the actual situation. To ensure this intended behavior, we modify the (first-order) definition of *Trans* for primitive actions as follows:

$$\begin{aligned}
&Trans(\alpha, s, \delta, s') \equiv \\
&\quad \neg online(s) \wedge Poss(\alpha[s], s) \wedge \delta = \mathsf{nil} \wedge s' = do(\alpha[s], s) \vee \\
&\quad online(s) \wedge \forall \tau. \alpha \neq waitFor(\tau) \wedge Poss(\alpha[s], s) \wedge \delta = \mathsf{nil} \wedge s' = do(\alpha[s], s) \vee \\
&\quad online(s) \wedge \exists \tau. \alpha = waitFor(\tau) \wedge \tau[s, start(s)] \wedge \delta = \mathsf{nil} \wedge s = s'.
\end{aligned}$$

If we are in projection mode, then in situation s there is a transition from the primitive or *waitFor* action α to a successor configuration $\langle \delta, s' \rangle$ if and only if α is possible, $s' = do(\alpha[s], s)$ and $\delta = \mathsf{nil}$, just as before (cf. Sections 3.2 and 4.2.1). The same holds if we are in on-line execution mode and a is not a *waitFor* action. If we are in on-line execution mode and a is a *waitFor*(τ) action, however, a is treated as a special kind of test: there is a transition if and only if the *t-form* τ is true at the beginning of the actual situation, that is if $\tau[s, start(s)]$ holds. In this case, the situation does not change along the transition, that is $s = s'$. As before, nothing remains to be done in the new configuration, thus $\delta = \mathsf{nil}$.

Similarly, the new second-order definition for *Trans* obtains by replacing the old assertion for primitive actions by the above definition, modulo textual substitution of *Trans* with T.

5.1.4 On-Line Execution Traces

Let us now consider how the actual on-line execution of a cc-Golog program works. The on-line interpreter searches for a next legal transition; if there is any such transition, it commits to it, and then continues. If the transition involves a *send* action, the interpreter checks whether this signals an activation of a low-level process and, as a side-effect, activates the actual low-level process if necessary.[2] Concurrently, the low-level execution system can cause an exogenous *reply* or *ccUpdate* action at any time. Thus, an on-line execution consists of a sequence of transitions caused by the high-level program, with an arbitrary number of exogenous actions between any two ordinary transitions. Formally, we define on-line execution as follows:[3]

Definition 7 *Let* AX *be the axioms for cc-Golog (including the new definition of Trans from Section 5.1.3, the axioms dealing with start, ccUpdate and online from Section 5.1.2 and the axioms needed for the encoding of cc-Golog programs as first-order terms) together with a situation calculus axiomatization of an application domain. Then an on-line execution with respect to* AX *of a program σ_0 in a situation s_0 is a sequence $\sigma_0, s_0, ..., \sigma_n, s_n$ such that for $i = 0, ...n-1$:*

1. $\mathrm{AX} \models \mathit{Trans}(\sigma_i, s_i, \sigma_{i+1}, s_{i+1})$; *or*
2. $\exists a, i, n.\ a = \mathit{reply}(i, n) \wedge \sigma_{i+1} = \sigma_i \wedge s_{i+1} = do(a, s_i)$; *or*
3. $\exists a, \vec{x}, t.\ a = \mathit{ccUpdate}(\vec{x}, t) \wedge [\forall \sigma', s'.\ \mathit{Trans}(\sigma_i, s_i, \sigma', s') \supset t \leq \mathit{start}(s')] \wedge \sigma_{i+1} = \sigma_i \wedge s_{i+1} = do(a, s_i)$.

We call an on-line execution completed *if* $\mathrm{AX} \models \mathit{Final}(\sigma_n, s_n)$. *Besides, we say that there is an on-line execution of σ_0 in s_0 that results in (σ_n, s_n) if and only if there is an on-line execution $\sigma_0, s_0, ..., \sigma_n, s_n$ of σ_0 in s_0. Finally, we say that a situation s_n is* a legal on-line execution trace of σ_0 in s_0 *if and only if there is a program σ_n such that there is an on-line execution of σ_0 in s_0 that results in (σ_n, s_n).*

By this definition, we only make the following assumption about the way in which exogenous *reply* and *ccUpdate* actions and actions performed by the high-level controller interleave: *if the high-level plan can perform an action at time t, and no exogenous action occurs at time t, then this action is executed by the high-level controller* (cf. Assertion 3). For example, if

[2] We will elaborate on this topic in Section 8.2, where we describe an implementation of a run-time system that couples cc-Golog to a real robot.

[3] This defintion is similar to the definition of on-line execution in [dGLS01].

the high-level plan is about to perform a *grabWheels*, then this action can be delayed by the occurrence of *reply* actions, but not by a *ccUpdate* that makes time advance. Otherwise, we do not impose any constraints on the occurrences of exogenous actions.

Note that (the first assertion of) this definition requires that there is a concrete transition to a *particular* configuration $\langle \sigma_{i+1}, s_{i+1} \rangle$ logically implied by the axioms, and not just the existence of a transition, i.e. $\exists \delta, s'.\mathit{Trans}(\sigma, s, \delta, s')$. This reflects the fact that the interpreter can only perform a transition involving the execution of an action if it is clear *which* action is to be executed. The following example illustrates the importance of this requirement. Let Γ be the axioms for cc-Golog together with a situation calculus theory involving the primitive action $\mathit{say}(\mathit{txt})$. Furthermore, let Γ be such that $\Gamma \not\models \phi[S_0]$ and $\Gamma \not\models \neg\phi[S_0]$, and $\Gamma \models \mathit{Poss}(\mathit{say}(\mathit{txt}), s)$. Consider the program $\sigma = \mathsf{if}(\phi, \mathit{say}(\mathit{its\_true}), \mathit{say}(\mathit{its\_false}))$. Then although there is a legal off-line execution of σ is S_0, i.e. $\Gamma \models \exists s'.\mathit{Do}(\sigma, S_0, s')$, there is no on-line execution of σ is S_0. This is because there is no particular transition logically implied by the axioms; neither $\Gamma \models \mathit{Trans}(\sigma, S_0, \mathsf{nil}, \mathit{do}(\mathit{say}(\mathit{its\_true}), S_0))$ nor $\Gamma \models \mathit{Trans}(\sigma, S_0, \mathsf{nil}, \mathit{do}(\mathit{say}(\mathit{its\_false}), S_0))$ holds.

5.1.5 Examples

To illustrate the concepts introduced in this section, let us now consider some on-line execution traces of the plan Π_{simple}, which was already introduced in Section 4.3.3.

$$\Pi_{\mathit{simple}} \doteq \mathsf{withPol}([\mathit{waitFor}(\mathit{inHallway}), \mathit{say}(\text{“in Hallway”})], \\ [\mathit{send}(\mathit{destRoom}, 6205), (\mathit{reg}(\mathit{arrivedAt}) = 6205)?])$$

As before, we assume that initially the robot is in Room 6213. Π_{simple}'s first action is $\mathit{send}(\mathit{destRoom}, 6205)$, which results in an activation of the navigation process, which we then assume to provide the high-level controller with *ccUpdate* actions every .25 sec. After 1 second, the execution results in situation S_{exec1}, where the p_i, q_i stand for appropriate $x-y$-coordinates along the path of the robot (we have left out the 3rd argument (voltage level) of *ccUpdate* for simplicity). S_{exec1} is visualized in Figure 5.2, where the black points represent the *ccUpdate*s.

$$S_{\mathit{exec1}} \doteq \mathit{do}([\mathit{send}(\mathit{destRoom}, 6205), \\ \mathit{ccUpdate}(p_1, q_1, 0.25), \mathit{ccUpdate}(p_2, q_2, 0.5), \\ \mathit{ccUpdate}(p_3, q_3, 0.75), \mathit{ccUpdate}(p_4, q_4, 1.0)], S_0)$$

It is not hard to see that S_{exec1} is a legal on-line execution trace of Π_{simple} in S_0. Let $\mathrm{AX}_{\mathrm{ccx}}$ be the new (second-order) axioms for cc-Golog together with the foundational axioms of the situation calculus from Section 3.1, axioms required for *t-form*'s, *val* axioms for the *t-function*s of the previous chapter, the precondition axiom for *waitFor*, the new definitions of this chapter concerning *start*, *online* and *ccUpdate*, the set of axioms $\mathrm{AX}_{\mathrm{arch}}$ from Section 4.3.1, the axioms needed for the encoding of cc-Golog programs as first-order terms and the fact $\mathit{online}(S_0)$. Finally, let Γ be $\mathrm{AX}_{\mathrm{ccx}}$ together with the new successor state axioms for *robotLoc* and *battLevel* from Section 5.1.1, initial state axioms stating that the robots initial position is *constant* and lies within Room 6213 and that the initial battery voltage is sufficiently high, and precondition axioms stating that *endGo* is always possible and *startGo* is only possible if the destination differs from the robot's current location. Then, there is an on-line execution of Π_{simple} in S_0 with respect to Γ that results in situation S_{exec1} with remaining program:

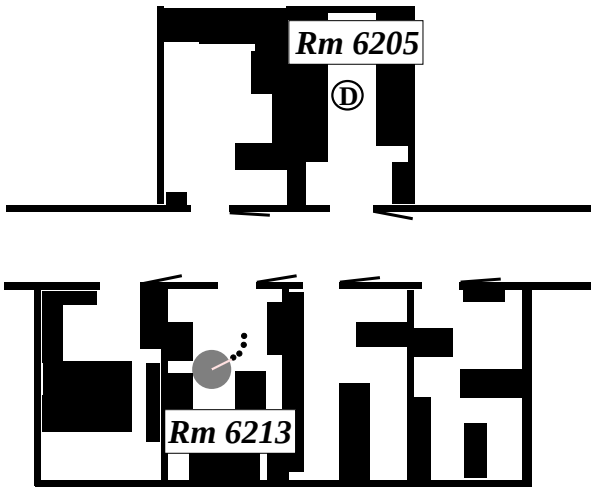


Figure 5.2: Execution Scenario of Π_{simple}

$$\delta = \mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ [\mathsf{nil}, reg(arrivedAt) = 6205]?).$$

Proof: (Sketch) The first transition results from the execution of a primitive action by Π_{simple} (here, by transition we mean that one of the three conditions of Definition 7 is satisfied). From the definition of *Trans*, together with the facts $Poss(send(destRoom, 6205), S_0)$ and $\neg inHallway[S_0, start(S_0)]$:

$$\begin{aligned} &Trans(\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ &\qquad [send(destRoom, 6205), (reg(arrivedAt) = 6205)?]), S_0, \delta, s') \equiv \\ &\quad s = do(send(destRoom, 6205), S_0) \wedge \\ &\quad \delta = \mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ &\qquad [\mathsf{nil}, (reg(arrivedAt) = 6205)?]). \end{aligned} \tag{5.1}$$

Let Π_1 and S_1 refer to the program respectively to the situation component of the new configuration. From the definition of *Trans*, together with the fact that neither $\exists t.inHallway[S_1, t]$ nor $reg(arrivedAt, S_1) = 6205$ holds, we obtain $\neg Trans(\Pi_1, S_1, \delta, s')$. Note however that $\langle \Pi_1, S_1 \rangle$ is not final. In fact, in this configuration the program is waiting until an exogenous action signals that the robot has either reached the hallway or is arrived in Room 6205.

Indeed, the other transitions leading to S_{exec1} are exogenous “*ccUpdate*” transitions. While they leave the program component unaffected, they subsequently modify the situation term from S_1 to

- $S_2 \doteq do([send(destRoom, 6205), ccUpdate(p_1, q_1, 0.25)], S_0)$;
- $S_3 \doteq do([send(destRoom, 6205), ccUpdate(p_1, q_1, 0.25), ccUpdate(p_2, q_2, 0.5)], S_0)$;
- $S_4 \doteq do([send(destRoom, 6205), ccUpdate(p_1, q_1, 0.25), \\ ccUpdate(p_2, q_2, 0.5), ccUpdate(p_3, q_3, 0.75)], S_0)$;
- $S_5 \doteq S_{exec1}$.

Let us now verify that the sequence $\sigma_0, S_0, ...\sigma_5, S_5$ with $\sigma_0 = \Pi_{simple}$ and $\sigma_i = \Pi_1$ for $i = 1, ..5$ satisfies Definition 7. The first transition is legal because of Π_{simple}'s initial transition (5.1). The other four transitions are "*ccUpdate*" transitions, which are allowed to make time advance because Π_1 is blocked, i.e. $\forall \delta', s'. \neg Trans(\sigma_1, S_1, \delta', s')$. □

Note that in the new configuration $\langle \Pi_1, S_{exec1} \rangle$, where Π_1 is what remains from the initial plan in S_{exec1}, the program Π_1 is still blocked. The on-line execution can only evolve as a result of further exogenous actions, which either signal that the robot has reached the hallway or has arrived in Room 6205.

Next, let us consider the situation S_{exec2} where the robot has just reached the hallway (we assume that $\langle p_8, q_8 \rangle$ is the first position within the hallway):

$$S_{exec2} \doteq do([ccUpdate(p_5, q_5, 1.25), ccUpdate(p_6, q_6, 1.5), \\ ccUpdate(p_7, q_7, 1.75), ccUpdate(p_8, q_8, 2.0)], S_{exec1}).$$

Just as before, it is possible to show that S_{exec2} is a legal on-line execution trace of Π_{simple} in S_0. In particular, there is an on-line execution of Π_{simple} in S_0 that results in S_{exec2} with remaining program:

$$\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ [\mathsf{nil}, reg(arrivedAt) = 6205]?).$$

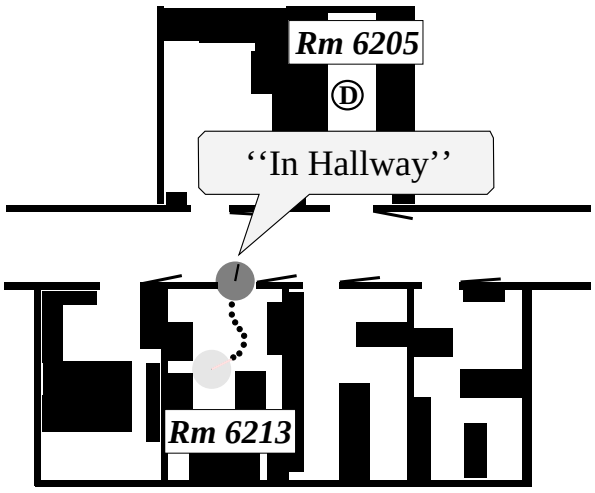


Figure 5.3: Execution Scenario of Π_{simple} (ii)

While so far the high-level program was blocked, in S_{exec2} the special test *waitFor*(*inHallway*) becomes true. Formally, it is possible to prove:

$$\Gamma \models Trans(\mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})], \\ [\mathsf{nil}, reg(arrivedAt) = 6205]?), S_{exec2}, \\ \mathsf{withPol}([\mathsf{nil}, say(\text{“in Hallway”})], \\ [\mathsf{nil}, reg(arrivedAt) = 6205]?), S_{exec2}).$$

Note that the situation component of the new configuration does not differ from its predecessor since *waitFor* is now merely a test. The next action to be executed by the high-level controller is *say*(“in Hallway”). Formally:

$$\Gamma \models \mathit{Trans}(\mathsf{withPol}([\mathsf{nil}, \mathit{say}(\text{“in Hallway”})], [\mathsf{nil}, \mathit{reg}(\mathit{arrivedAt}) = 6205]?), S_{exec2}, \mathsf{withPol}(\mathsf{nil}, [\mathsf{nil}, \mathit{reg}(\mathit{arrivedAt}) = 6205]?), \mathit{do}(\mathit{say}(\text{“in Hallway”}), S_{exec2})).$$

The resulting situation $S_{exec3} \doteq \mathit{do}(\mathit{say}(\text{“in Hallway”}), S_{exec2})$ is visualised in Figure 5.3. Just as before, it is possible to show that it is a legal on-line execution trace of Π_{simple} in S_0.

5.2 Interleaving Projection and On-Line Execution

In [dGL99b], de Giacomo and Levesque suggest that it is often useful to interleave on-line execution and projection. With our model of time we can take this idea one step further and define projection during on-line execution with the possibility to explicitly put a time-bound on the projection. A prerequisite for the bounded projection of plans during on-line execution is the ability to project plans in non-initial situations.

5.2.1 Projection in Non-Initial Situations

While so far we have only considered the projection of cc-Golog plans in the initial situation, a robot that is to exhibit intelligent behavior over an extended period of time must be able to deduce the effects of the remaining plan (as well as of different plans) in non-initial situations. To see how projection works in non-initial situations, let us again consider our example scenario, where projection makes use of a model of the navigation process.

Suppose we are in Situation S_{exec1} illustrated in Figure 5.2 (page 87), which occurs during the on-line execution of Π_{simple}. First, let us consider the value of the fluents *robotLoc* and $\mathit{reg}(\mathit{destRoom})$ in S_{exec1}. Let Γ be defined as before. It is easy to see that:

$$\Gamma \models \mathit{robotLoc}(S_{exec1}) = \mathit{constant}(p_4, q_4) \wedge \mathit{reg}(\mathit{destRoom}, S_{exec1}) = 6205.$$

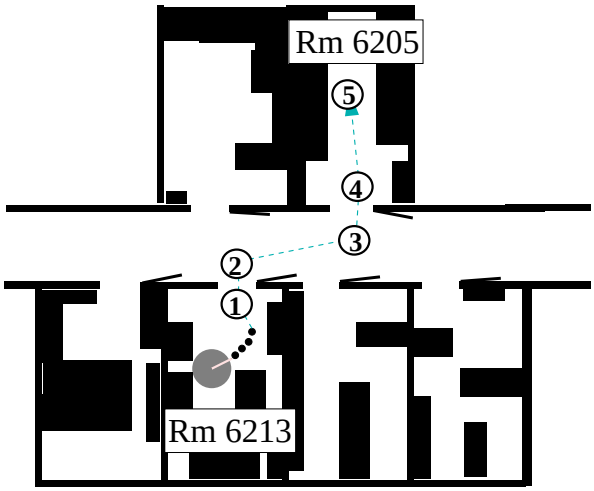


Figure 5.4: Projection during Execution

Given the updated value of *robotLoc*, we can now correctly project that the remaining plan will cause the robot to travel directly to its destination. In particular, due to the fact

that $reg(destRoom) = 6205$, the procedure *navProc* will immediately become active because the robot actually is not inside Room 6205. Note the use of *clipOnline*, which forces a switch to projection mode. The following projection is visualized in Figure 5.4.

$$\begin{aligned}
\Gamma \models [&Proj(do(clipOnline, S_{exec1}),\\
&\quad \mathsf{withPol}([waitFor(inHallway), say(\text{“in Hallway”})],\\
&\qquad [\mathsf{nil}, reg(arrivedAt) = 6205]?),\\
&\quad navProc, s)] \equiv\\
&[s = do([startGo(l_1),\\
&\qquad waitFor(near(l_1)), startGo(l_2),\\
&\qquad waitFor(inHallway), say(\text{“in Hallway”})\\
&\qquad waitFor(near(l_2)), startGo(l_3),\\
&\qquad waitFor(near(l_3)), startGo(l_4),\\
&\qquad waitFor(near(l_4)), startGo(l_5),\\
&\qquad waitFor(near(l_5)), endGo,\\
&\qquad reply(arrivedAt, 6205)], do(clipOnline, S_{exec1}))]
\end{aligned}$$

The above can be shown by similar reasoning as in Section 4.3.3, where we considered the projection of Π_{simple} in S_0.

Similarly, in situation $S_{exec3} \doteq do(say(\text{“In Hallway”}), S_{exec2})$ we can project the remaining program $\mathsf{withPol}(\mathsf{nil}, [\mathsf{nil}, reg(arrivedAt) = 6205]?)$. Note that in the following projection, which is visualized in Figure 5.5, the robot directly travels towards l_3 without first visiting l_2. This is because in S_{exec3} the robot is no longer in Room 6213, and thus the conditional in the procedure *travelTo* is not executed (see Section 4.3.2, page 72).

$$\begin{aligned}
\Gamma \models &Proj(do(clipOnline, S_{exec3}),\\
&\quad \mathsf{withPol}(\mathsf{nil}, [\mathsf{nil}, reg(arrivedAt) = 6205]?), navProc, s) \equiv\\
&s = do([startGo(l_3),\\
&\qquad waitFor(near(l_3)), startGo(l_4),\\
&\qquad waitFor(near(l_4)), startGo(l_5),\\
&\qquad waitFor(near(l_5)), endGo,\\
&\qquad reply(arrivedAt, 6205)], do(clipOnline, S_{exec3}))]
\end{aligned}$$

The reason why projections in non-initial situations of the example scenario are fairly straightforward is that we never have to remember the state the actual navigation process is in. More precisely, for any given position, the cc-Golog model of the navigation process yields an appropriate approximation of the remaining trajectory of the robot (as illustrated by the above projections). While many low-level processes used in mobile robots seem to have this property, we remark that in Section 7.3.3 we will show how processes can be dealt with where one needs to keep track of their internal state during execution.

5.2.2 (Limited) Lookahead: Projection Tests

Motivated by the work on incremental execution of [dGL99b], which suggest that a combination between on-line execution and projection (which they call off-line execution) arises as a practical and still powerfull scheme of execution, we add a local lookahead constructor which allows projection under user control, with the possibility to explicitly put a time-bound on the projection. Formally, we define the predicate $Lookahead(\phi, t, \sigma, ll_{model}, s)$, which is true, roughly, if ϕ holds at time t in the projected execution of a plan σ in situation s. Here, ϕ

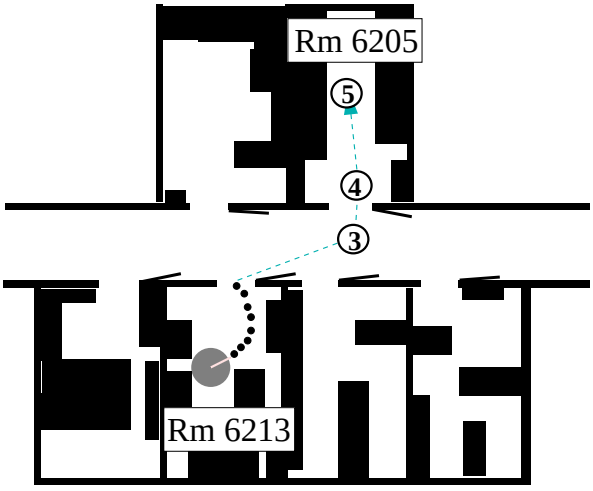


Figure 5.5: Projection during Execution (ii)

stands for a situation calculus formula where the special situation constant *now* may be used to refer to the resulting projected situation. As before, ll_{model} refers to an appropriate model of the low-level processes.

$$
\begin{aligned}
&\mathit{Lookahead}(\phi, t, \sigma, ll_{model}, s) \equiv \\
&\quad \exists \sigma^*, s^*.\phi[s^*] \wedge \mathit{start}(s^*) \leq t \wedge \\
&\quad \mathit{Trans}^*(\mathsf{withPol}(ll_{model}, \sigma), \mathit{do}(\mathit{clipOnline}, s), \sigma^*, s^*) \wedge \\
&\quad [\forall \sigma^{**}, s^{**}.\mathit{Trans}^*(\sigma^*, s^*, \sigma^{**}, s^{**}) \supset \mathit{start}(s^{**}) > t \vee \mathit{Final}(\sigma^*, s^*)]
\end{aligned}
$$

Again, we use *clipOnline* to switch to projection mode. The disjunct involving *Final* covers the case where the projected plan ends before time t. In order to allow local lookahead within cc-Golog programs, we allow the use of the *term* $\mathsf{Lookahead}(\phi, t, \sigma, ll_{model})$ within test conditions. Lookahead is a reified version of *Lookahead* with situation argument suppressed, with the idea that during execution Lookahead conditions are interpreted with respect to the actual situation (just like reified fluents).

Technically, the use of projection tests within cc-Golog programs raises subtle issues. First, we have to reify them as first-order terms, just like any fluent which may be used within programs. Second, in reifying *Lookahead* great care has to be taken to avoid defining self-referencing sentences, like cc-Golog programs appealing to their own effects. Here, we will not go into the details of reifying *Lookahead* within the language except to note that running into self-referencing programs is avoided by distinguishing between two sorts of cc-Golog programs: "ordinary" cc-Golog programs, and cc-Golog programs including projection tests. Only the former are allowed to occur within Lookaheads. See Appendix A.2.2 for an in-depth treatment of these issues.

5.2.3 Projection Tests at Work

Using Lookahead within test conditions, a cc-Golog plan can check whether a possible behavior would lead to certain conditions, and thus deliberate over different possible subplans. In addition, the lookahead can be limited to a certain amount of time t, which seems very

useful in order to make quick decisions. A possible application of this test, illustrated by the following program, would be to check if the battery level is going to drop below 46V in the next 300 seconds if the robot comes close to the battery docking station:

$$\begin{array}{l}\mathsf{proc}(\Pi_{lookahead},\\ \quad \mathsf{withPol}(\mathsf{whenever}(nearDockingStation,\\ \qquad \mathsf{if}(\mathsf{Lookahead}(lowBattLevel, start(now)+300, deliverMail, navProc),\\ \qquad\quad [grabWheels, chargeBatteries, releaseWheels])),\\ \qquad \mathsf{withCtrl}(wheels, deliverMail))).\end{array}$$

Here, we use *lowBattLevel* as an abbreviation for an expression that is true if and only if the value of *battLevel* is below $46V$ in the projected execution trace.

Let us now consider an on-line execution of $\Pi_{lookahead}$, assuming that the robot's initial position is in Room 6213, that the battery docking station is in Room 6204, and that there are two letters to be delivered, one to Room 6205 and one to Room 6214, just as in the example illustrated in Figure 4.5 on page 78. This time however, the robot does not wait until the battery level drops below 46V before it reacts, but instead makes use of the lookahead construct. The resulting on-line execution trace is visualized in Figure 5.6. We remark that the behavior exhibited by the high-level plan making use of a projection test is superior to the behavior illustrated in Figure 4.5; in particular, the length of the overall trajectory is shorter.

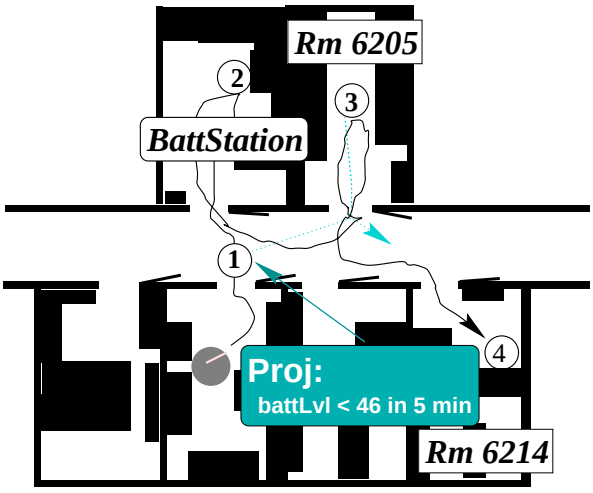


Figure 5.6: Delivery with lookahead

Let us now take a closer look at the resulting on-line execution trace. Up to the node marked "1", the execution runs in a similar way as the previous examples. Let $S_{lookahead}$ refer to the situation where *nearDockingStation* becomes true, that is to the situation where the robot has just reached node "1". Furthermore, let us assume that in $S_{lookahead}$ the battery level is quite low (recall that the actual voltage level is provided by *ccUpdate* actions; see Section 5.1.1). At this point, the high-level controller *running in on-line execution mode* will evaluate $Lookahead(lowBattLevel, start(now)+300, deliverMail, navProc)$, that is, it will *project* the effects of *deliverMail*. The projection test will look for a successor configuration $\langle \sigma^*, s^* \rangle$ whose *start* time lies just below $start(S_{lookahead})+300$, and will then verify that in

this configuration the battery level is below 46 V. Formally, this means that the projection test will verify:

$$\begin{aligned}\Gamma \models \exists \sigma^*, s^*.\, &Trans^*(\mathsf{withPol}(navProc, deliverMail), do(clipOnline, S_{lookahead}), \sigma^*, s^*) \wedge \\ &[\forall \sigma^{**}, s^{**}.\, Trans^*(\sigma^*, s^*, \sigma^{**}, s^{**}) \supset start(s^{**}) > start(S_{lookahead}) + 300 \vee \\ &\quad Final(\sigma^*, s^*)] \wedge \\ &start(s^*) \leq start(S_{lookahead}) + 300 \wedge \\ &[battLevel \leq 46][s^*, start(S_{lookahead}) + 300].\end{aligned}$$

Assuming that the above holds, the high-level controller then interrupts *deliverMail* in $S_{lookahead}$ to first charge its batteries (node "2"). Thereafter, it completes the delivery.

Another possible application of the Lookahead construct arises in the context of multi-robot control (cf. [ACF+98, AFH+98]). Suppose a user makes a request to have a letter delivered by one of several courier robots. In order to determine which robot should deliver the letter, each robot might use projection to determine the cost that would arise if it were to carry out this job, and the task could then be assigned to the robot with a minimal cost estimate.

5.3 Discussion

In summary, we have extended cc-Golog so that it becomes suitable for both projections of plans and their on-line execution. To account for the fact that time advances differently in projections and during on-line execution, we explicitly distinguished between the two modes of operation. In particular, during on-line execution we make use of exogenous *ccUpdate* actions to frequently update the value of continuous fluents like *robotLoc*, and treat *waitFor* instructions simply as a special kind of test. Finally, we have shown how to interleave on-line execution and a form of time-bounded projection.

As mentioned before, the idea of interleaving projection and on-line execution was first explored by de Giacomo and Levesque [dGL99b]. However, they consider neither time nor the idea of low-level processes which interact with a high-level controller in complex ways. Most importantly, there is no distinction between the effects on fluents during projection and on-line execution, a distinction we feel is necessary when it comes to modeling essential features of mobile robots such as their location at a given time. Lespérance and Ng [LN00] have extended de Giacomo and Levesque's ideas in a direction which bears some resemblances to ours. They propose that during projection one also needs to consider a *simulated environment* which, for example, produces exogenous actions to inform the high-level controller that the destination is reached. However, their notion of a simulated environment remains fairly simple since they are not able to model temporally extended processes, and they also do not distinguish between the effects on fluents during projection and on-line execution.

While the approach of [dGL99b] provides nondeterministic instructions, in cc-Golog deliberation over different sub-plans is only possible by means of explicit projection tests. This scheme is certainly neither as elegant nor as expressive as the use of nondeterministic instructions. However, the explicit use of constructs like Lookahead gives the user finer-grained control over the amount of projection involved in the execution of a plan, which is useful when dealing with limited resources. Finally, using projection tests it is possible to specify a program which first projects the successful execution of a program P, but then executes a *different* program P'. This can be quite useful, as illustrated by the multi-robot example,

where (potentially) every robot will project that it can carry out the next job, but only one robot will actually do so.

The idea to distinguish between the plan time effects and the run time effects of an action is also present in [BP98], where both the plan time effects and the run time effects of an action on the knowledge of an agent is modeled. As for the *ccUpdate* actions, we remark that they represent a kind of "passive" sensing, where sensory information continually becomes available without the need for an explicit request, as opposed to more classical approaches to sensing like [GW96, Lev96, WAS98]. This view of sensing is somewhat similar to that of Poole [Poo96, Poo98], and de Giacomo and Levesque [dGL99a], where *sensing functions* are used to represent on-board sensors that provide sensor readings at any time. However, while in [dGL99a] projection is defined in terms of *histories*, which intuitively consist of a situation *and* a formula representing the values of the sensors in each situation, we manage solely with a situation enriched with an arbitrary number of *ccUpdate* actions.

Chapter 6

pGOLOG - Dealing with Probabilistic Uncertainty

In the previous chapters, we have assumed that all low-level processes have deterministic effects. In particular, there is no uncertainty about whether or not a low-level process achieves the desired results. However, an important feature of real robot environments is the inherent uncertainty in what the world is like and the outcome of many of a robot's low-level processes, due to the fact that robot hard- and software is imperfect and error-prone. For example, if a robot tries to pickup a cup, many different outcomes are possible: the robot may completely miss the cup, the cup may drop on the floor, the robot may push adjacent objects or might even break the cup or an adjacent object. These outcomes typically occur with different probabilities. For example, the pickup may succeed perfectly about 80% of the time and may have some other outcomes with lower probability.

In this chapter, we will show how the GOLOG framework presented so far can be extended to allow the projection of high-level plans interacting with noisy low-level processes, based on a probabilistic characterization of the robot's beliefs. In particular, we model low-level processes with uncertain outcome as probabilistic programs in a probabilistic extension of GOLOG which we call pGOLOG. The intuition is that the different probabilistic branches of the programs correspond to different outcomes of the processes. Given a faithful characterization of the low-level processes in terms of pGOLOG programs, we can then reason about the effects of their activation through simulation of the corresponding pGOLOG model. As a result of this probabilistic setting, projection now yields the *probability* of a plan to achieve a goal, which leads us to the notion of *probabilistic projection*.

To get a better feel for what we are aiming at, let us consider the following *ship/reject*-example, adapted from [DHW94]: We are given a manufacturing robot with the goal of having a widget painted (PA) and processed (PR). Processing widgets is accomplished by rejecting parts that are flawed (FL) or shipping parts that are not flawed. This can be done by means of the low-level processes *ship* and *reject*. The robot also has a low-level process *paint* that usually makes PA true. Initially, all widgets are flawed if and only if they are blemished (BL), and the probability of being flawed is 0.3.

Although the robot cannot tell directly if the widget is flawed, the low-level process *inspect* can be used to determine whether or not it is blemished. *inspect* reports $\overline{OK}$ ("not OK") if the widget is blemished, and OK if not. The *inspect* process can be used to decide whether or not the widget is flawed because FL and BL are initially perfectly correlated. The use

of inspect is complicated by two things, however. (1) *inspect* is not perfect: if the widget is blemished, then 90% of the time it reports $\overline{OK}$, but 10% of the time it erroneously reports *OK*. If the widget is not blemished, however, *inspect* always correctly reports *OK*. (2) Painting the widget removes a blemish but not a flaw, so executing *inspect* after the widget has been painted no longer conveys information about whether it is flawed. All low-level processes can always be activated, but may result in different effects. *paint* makes *PA* TRUE (and *BL* FALSE) with probability 0.95 if the widget was not already processed. Otherwise, it causes an execution error (*ER*). *ship* and *reject* always make *PR* TRUE, *ship* makes *ER* true if *FL* holds, and *reject* makes *ER* TRUE if *FL* does not hold. The goal of our manufacturing robot is to have the widget painted and processed with at least 90% probability.

In this scenario, an example projection task is the following: how probable is it that the plan "first *inspect*, then *paint* the widget; afterwards, if *OK* holds then *ship* else *reject* it" will falsely ship a flawed widget? In order to answer this kind of question, we have to represent the robot's uncertainty about the initial state of the world, and must reason about low-level processes like *paint* which have uncertain outcomes. As mentioned above, we make use of probabilistic pGOLOG programs to model noisy low-level processes; for example, we model *paint* by a program which either causes *PA* to become true, or not. Besides, we have to deal with low-level processes like *inspect* which provide (noisy) information about the state of the world, i.e. we have to integrate *sensing* into our architecture. We call low-level processes like *inspect* "sensor processes." Roughly, we deal with sensor processes as follows: during actual execution, we treat answers like *OK* as special exogenous actions. For the task of projection, on the other hand, we model the sensor processes by pGOLOG programs involving these special actions.

As for the representation of the uncertainty about the state of the world, we follow Bacchus, Halpern and Levesque [BHL99] and characterize the robot's epistemic state by a distribution over possible situations considered possible. While in general this probabilistic representation of the robot's beliefs would allow us to specify GOLOG-style plans whose tests and conditionals appeal to the robot's real-valued beliefs, for means of simplicity in this chapter we do not allow plans involving conditionals like "if the robot's belief that the widget is flawed is beyond 10%, then reject the widget." Instead, we only consider plans whose actions are essentially conditioned on the answers provided by the sensor processes, like "*OK*" or "$\overline{OK}$," which we call *directly observable*. We will consider the issue of real-valued belief tests in the next chapter.

This chapter is organized as follows. In the next section, we define pGOLOG and show how low-level processes with uncertain outcome like *paint* can be modeled in pGOLOG. In Section 6.2, we extend the robot control architecture used in the previous chapters to account for uncertainty and sensing. Finally, in Section 6.3 we show how probabilistic projection works in pGOLOG, and briefly discuss the relation between probabilistic projection and the concept of *expected utility*.

6.1 pGOLOG: a Probabilistic GOLOG Dialect

As the above example illustrates, it is often convenient to model real-world processes as having different possible probabilistic outcomes. For example, the *paint* process might not be guaranteed to succeed, and this inaccuracy can naturally be modeled by assigning probabilities to different outcomes (e.g. success and failure). To take this into account, we introduce a new

sort *probability* ranging over the subset $[0, 1]$ of the reals[1] and extend cc-Golog with a new probabilistic branching instruction: $\mathsf{prob}(p, \sigma_1, \sigma_2)$. Here, σ_1 and σ_2 are pGOLOG programs and p is a probability which lies between 0 and 1, i.e. $0 < p < 1$. The intended meaning of $\mathsf{prob}(p, \sigma_1, \sigma_2)$ is to execute program σ_1 with probability p, and σ_2 with probability $1-p$. The idea is to model noisy low-level processes through probabilistic programs, where the different probabilistic branches of the programs correspond to different outcomes of the processes. Altogether, the new language pGOLOG offers the following instructions:

α	primitive action or *waitFor*
$\phi?$	wait/test action
$[\sigma_1, \sigma_2]$	sequence
$\mathsf{if}(\phi, \sigma_1, \sigma_2)$	conditional
$\mathsf{while}(\phi, \sigma)$	loop
$\mathsf{withCtrl}(\phi, \sigma)$	guarded execution
$\mathsf{conc}(\sigma_1, \sigma_2)$	prioritized execution
$\mathsf{prob}(p, \sigma_1, \sigma_2)$	probabilistic execution of either σ_1 or σ_2
$\{\mathsf{proc}(P_1(\vec{v_1}), \beta_1); ...; \mathsf{proc}(P_n(\vec{v_n}), \beta_n); \sigma\}$	procedures.

To illustrate the use of pGOLOG, we will now model the effects of the *paint* process by the pGOLOG program *paintProc*. Intuitively, if the widget is already processed, trying to *paint* it results in an error. Otherwise, *paint* will result in the widget being painted with probability 95%. There is also a 5% chance that although the widget is not processed, *paint* will remain effectless. In both probabilistic cases, a possible blemish is removed. To model the effects of *paint*, we make use of the fluents *PA*, *FL*, *BL*, *PR* and *ER* to represent the properties of our example domain, and assume successor state axioms that ensure that the truth value of *PA* is only affected by the primitive actions *setPA* and *clipPA*, whose effect is to make it TRUE respectively FALSE. Similarly for the other fluents. Here and in the remainder of this thesis, we use $\mathsf{prob}(p, \sigma)$ as a shorthand for $\mathsf{prob}(p, \sigma, \mathsf{nil})$.

$$paintProc \doteq \mathsf{if}(PR, setER, [clipBL, \mathsf{prob}(0.95, setPA)])$$

Given this characterization of the paint process, we can reason about the effects of its activation through simulation of the pGOLOG model. We stress that pGOLOG programs like the above are not intended for actual execution. The purpose of pGOLOG programs like *paintProc* is to model the behavior of low-level processes as probabilistic programs, and thus to provide a faithful characterization of their effects on the world, similarly to the cc-Golog program *navProc* used in Section 4.3.2 as model of the navigation process.

6.1.1 A Weighted Transition Semantics

Similar to ConGolog, the semantics of pGOLOG is defined using a transition semantics, specifying which configuration $\langle \delta, s' \rangle$ can be reached from a configuration $\langle \sigma, s \rangle$ by a single step of computation. As in ConGolog, the use of transition semantics necessitates the reification of programs as first order terms in the logical language; see Appendix A. However, unlike in the case of cc-Golog and deterministic ConGolog, when dealing with probabilistic pGOLOG programs a configuration may have more than one successor configuration, and the different

[1] As before, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

successor configurations have a specific *probability*.[2] In particular, a configuration involving $\mathsf{prob}(p, \sigma_1, \sigma_2)$ has two possible successor configurations, which intuitively correspond to the execution of σ_1 respectively of σ_2. The respective probabilities of these successor configurations is p and $1 - p$.

In order to account for the different probabilities of the possible successor configurations, we replace the predicate *Trans* used in the definition of the semantics of ConGolog by the function *transPr* which takes the role of *Trans* but additionally assigns a degree of likelihood to different possible transitions. Roughly, $transPr(\sigma, s, \delta, s')$ is the transition probability associated with a given program σ and situation s as well as a new situation s' that results from executing σ's first primitive action in s, and a new program δ that represents what remains of σ after having performed that action. For simplicity, we first present a first-order definition of *transPr* that does not account for procedures. Furthermore, for readability we write

$$f(\vec{x}) = \textbf{ if } \exists \vec{y}.\phi(\vec{x}, \vec{y}) \textbf{ then } g(\vec{x}, \vec{y}) \textbf{ else } h(\vec{x})$$

as an abbreviation for

$$f(\vec{x}) = p \equiv \exists \vec{y}.\phi(\vec{x}, \vec{y}) \wedge p = g(\vec{x}, \vec{y}) \vee \neg\exists \vec{y}.\phi(\vec{x}, \vec{y}) \wedge p = h(\vec{x}),$$

where $\phi(\vec{x}, \vec{y})$ is a first-order formula with free variables among $\vec{x} \cup \vec{y}$, and similarly $g(\vec{x}, \vec{y})$ and $h(\vec{x})$ are functions whose arguments range over $\vec{x} \cup \vec{y}$ and $\vec{x}$, respectively. Similarly, the nested **if - then - else** construct

$$\begin{aligned} f(\vec{x}) = \textbf{ if } \exists \vec{y_1}.\phi_1(\vec{x}, \vec{y_1}) \textbf{ then } & g_1(\vec{x}, \vec{y_1}) \\ \textbf{else if } \exists \vec{y_2}.\phi_2(\vec{x}, \vec{y_2}) \textbf{ then } & g_2(\vec{x}, \vec{y_2}) \\ \textbf{else } & h(\vec{x}) \end{aligned}$$

is an abbreviation for

$$\begin{aligned} f(\vec{x}) = p \equiv & \exists \vec{y_1}.\phi_1(\vec{x}, \vec{y_1}) \wedge p = g_1(\vec{x}, \vec{y_1}) \vee \exists \vec{y_2}.\phi_2(\vec{x}, \vec{y_2}) \wedge p = g_2(\vec{x}, \vec{y_2}) \vee \\ & \neg[\exists \vec{y_1}.\phi_1(\vec{x}, \vec{y_1}) \vee \exists \vec{y_2}.\phi_2(\vec{x}, \vec{y_2})] \wedge p = h(\vec{x}). \end{aligned}$$

Formally, *transPr* is defined as follows:

$$transPr(nil, s, \delta, s') = 0$$

$$\begin{aligned} & transPr(\alpha, s, \delta, s') = \\ & \quad \textbf{if } Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s) \textbf{ then } 1 \textbf{ else } 0 \end{aligned}$$

$$\begin{aligned} & transPr(\phi?, s, \delta, s') = \\ & \quad \textbf{if } \phi[s] \wedge \delta = nil \wedge s' = s \textbf{ then } 1 \textbf{ else } 0 \end{aligned}$$

$$\begin{aligned} & transPr(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') = \\ & \quad \textbf{if } \phi[s] \textbf{ then } transPr(\sigma_1, s, \delta, s') \textbf{ else } transPr(\sigma_2, s, \delta, s') \end{aligned}$$

$$\begin{aligned} & transPr([\sigma_1, \sigma_2], s, \delta, s') = \\ & \quad \textbf{if } \exists \gamma.\delta = [\gamma, \sigma_2] \wedge transPr(\sigma_1, s, \gamma, s') > 0 \textbf{ then } transPr(\sigma_1, s, \gamma, s') \\ & \quad \textbf{else if } Final(\sigma_1, s) \wedge transPr(\sigma_2, s, \delta, s') > 0 \textbf{ then } transPr(\sigma_2, s, \delta, s') \textbf{ else } 0 \end{aligned}$$

$$\begin{aligned} & transPr(\mathsf{while}(\phi, \sigma), s, \delta, s') = \end{aligned}$$

[2] Of course, nondeterministic ConGolog programs [dGLL00] are also concerned with multiple successor configurations; however, nondeterministic programs do not consider the probability of a transition. We will elaborate on this topic in Section 6.4.

$$
\begin{array}{l}
\quad \textbf{if } \exists\gamma.\phi[s] \wedge \delta = [\gamma, \mathit{while}(\phi,\sigma)] \textbf{ then } \mathit{transPr}(\sigma, s, \gamma, s') \textbf{ else } 0 \\
\mathit{transPr}(\mathsf{withCtrl}(\phi,\sigma), s, \delta, s') = \\
\quad \textbf{if } \exists\gamma.\phi[s] \wedge \delta = \mathsf{withCtrl}(\phi,\gamma) \textbf{ then } \mathit{transPr}(\sigma, s, \gamma, s') \textbf{ else } 0 \\
\mathit{transPr}(\mathsf{conc}(\sigma_1,\sigma_2), s, \delta, s') = \\
\quad \textbf{if } \neg\mathit{Final}(\sigma_1, s) \wedge \neg\mathit{Final}(\sigma_2, s) \wedge \\
\qquad \exists\delta_1.\mathit{transPr}(\sigma_1, s, \delta_1, s') > 0 \wedge \delta = \mathsf{conc}(\delta_1, \sigma_2) \wedge \\
\qquad\quad \forall\delta_2, s_2.\mathit{transPr}(\sigma_2, s, \delta_2, s_2) > 0 \supset \mathit{start}(s') \leq \mathit{start}(s_2) \\
\quad \textbf{then } \mathit{transPr}(\sigma_1, s, \delta_1, s') \\
\quad \textbf{else if } \neg\mathit{Final}(\sigma_1, s) \wedge \neg\mathit{Final}(\sigma_2, s) \wedge \\
\qquad \exists\delta_2.\mathit{transPr}(\sigma_2, s, \delta_2, s') > 0 \wedge \delta = \mathsf{conc}(\sigma_1, \delta_2) \wedge \\
\qquad\quad \forall\delta_1, s_1.\mathit{transPr}(\sigma_1, s, \delta_1, s_1) > 0 \supset \mathit{start}(s') < \mathit{start}(s_1) \\
\qquad \textbf{then } \mathit{transPr}(\sigma_2, s, \delta_2, s') \textbf{ else } 0 \\
\mathit{transPr}(\mathsf{prob}(p,\sigma_1,\sigma_2), s, \delta, s') = \\
\quad \textbf{if } \delta = \sigma_1 \wedge s' = \mathit{do}(\mathit{tossHead}, s) \textbf{ then } p \\
\quad \textbf{else if } \delta = \sigma_2 \wedge s' = \mathit{do}(\mathit{tossTail}, s) \textbf{ then } 1-p \textbf{ else } 0
\end{array}
$$

Concerning the instructions already present in cc-Golog, the definition of *transPr* essentially reflects cc-Golog's (first-order) definition of *Trans*. In particular, by inspection it is easy to see that for non-probabilistic programs σ and δ the following holds:

$$\mathit{transPr}(\sigma, s, \delta, s') > 0 \text{ if and only if } \mathit{Trans}(\sigma, s, \delta, s').$$

As in cc-Golog, a program that consists of a single atomic action α results in the execution of $\alpha[s]$ and an empty remaining program with probability 1 if and only if $\alpha[s]$ is executable; all other configurations $\langle\delta, s'\rangle$ have probability 0. Similarly, a test can only result in a transition if ϕ holds in s. The execution of the conditional $\mathsf{if}(\phi,\sigma_1,\sigma_2)$ in s corresponds to the execution of σ_1 or σ_2, depending on the truth value of ϕ in s. The execution of $[\sigma_1,\sigma_2]$ in s may result in any successor situation that could be reached by the execution of σ_1, with a remaining program $[\gamma,\sigma_2]$, where γ is what remains of σ_1; or, if σ_1 is final, it just corresponds to the execution of σ_2. A $\mathsf{while}(\phi,\sigma)$ loop may only cause a transition if ϕ holds in s. A $\mathsf{withCtrl}(\phi,\sigma)$ construct is blocked if ϕ is false, and else behaves like σ. Finally, the concurrent execution of σ_1 and σ_2 means that one action of one of the programs is performed, whereby actions which can be executed earlier are always preferred. If both σ_1 and σ_2 are about to execute an action at the same time, then σ_1 which has higher priority takes precedence.

Let us now turn to the new instruction prob. The execution of $\mathsf{prob}(p,\sigma_1,\sigma_2)$ results in the execution of a dummy action *tossHead* or *tossTail* with probability p respectively $1-p$ with remaining program σ_1, respectively σ_2. The *tossHead* respectively *tossTail* actions ensure that the situation component of the two possible successor configurations differ. Intuitively, this is necessary because nothing prevents us from specifying a prob instruction with two identical programs σ_1 and σ_2, and if we would not distinguish the two possible resulting configurations by means of *tossHead* respectively *tossTail*, this would result in two identical successor configurations, that is, a single successor configuration which is assigned the probability p and $1-p$ at the same time, obviously a contradiction. Note that even though we could syntactically restrict the use of prob to different programs σ_i, the execution of these programs could still result in an identical execution trace s'', which would again result in a contradiction. We assume that *tossHead* and *tossTail* have no effects and are always possible.

As in Section 3.2, we also have to define the final configurations $\langle \sigma, s \rangle$. For all instructions already present in cc-Golog, the definition of *Final* remains unchanged. A prob instruction is never final:

$$Final(\mathsf{prob}(p, \sigma_1, \sigma_2), s) \equiv \text{FALSE}.$$

So far, we have only defined which successor configurations can be reached through a single transition. Now, as in the definition of the semantics of ConGolog we want to define which configurations can be reached by a *sequence of transitions*, that is we want to define the transitive closure of *transPr*. The definition of $transPr^*(\sigma, s, \delta, s')$, the transitive closure of *transPr*, is related to that of $Trans^*$. However, it is somewhat more complex because it does not only specify which configurations $\langle \delta, s' \rangle$ can be reached but also specifies the probability to reach them:

$$\begin{aligned} transPr^*(\sigma, s, \delta, s') = p \equiv & \forall t[... \supset t(\sigma, s, \delta, s') = p] \vee \\ & p = 0 \wedge \neg \exists p'. \forall t[... \supset t(\sigma, s, \delta, s') = p'] \end{aligned}$$

where the ellipsis stands for the universal closure of the conjunction of the following formulas:

$$t(\sigma, s, \sigma, s) = 1 \tag{6.1}$$

$$\begin{aligned} & [t(\sigma, s, \sigma^*, s^*) = p_2 \wedge transPr(\sigma^*, s^*, \delta, s') = p_1 \wedge p_1 > 0 \wedge p_2 > 0] \\ & \quad \supset t(\sigma, s, \delta, s') = p_1 * p_2. \end{aligned} \tag{6.2}$$

Basically, this formula says that i) if there is a path of nonzero transitions from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$, then $transPr^*(\sigma, s, \delta, s')$ is equal to the product of the transition probabilities p along this path (which we call its weight), otherwise it is zero; and ii) there are no two paths from one configuration to another with different weights (else, $transPr^*(\sigma, s, \delta, s')$ would have more than one value).

If there is a path of nonzero transitions, then (i) obtains, roughly, by "iterating" through Formula 6.2, making use of the reflexivity of t (Formula 6.1) for the case where there is a direct transition (i.e. a *transPr* connection) from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$. In this case, $\langle \sigma, s \rangle$ and $\langle \sigma^*, s^* \rangle$ are the same and the weight of $transPr^*$ corresponds to that of *transPr*. If there is no path without nonzero transitions, then one can always find a function t_1 which satisfies the ellipsis such that $t_1(\sigma, s, \delta, s') = 0$. Hence $transPr^*(\sigma, s, \delta, s') = 0$.

To see why ii) holds, let us assume that there are two paths with different weights from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$. Then no function t exists that satisfies Formula 6.2; therefore $\forall t[...]$ is vacuously TRUE, and $transPr^*(\sigma, s, \delta, s') = p$ for all p, a contradiction. Recall that to prevent this from happening when executing a *prob* even if $\sigma_1 = \sigma_2$, we introduced the dummy actions *tossHead* and *tossTail* which ensure that the situations associated with σ_1 and σ_2 are different. Formally, we have the following proposition:

Proposition 8: *Let* AX *be the foundational axioms of the epistemic situation calculus together with the definitions of transPr, Final and* $transPr^*$ *of this section plus the axioms needed for the encoding of* pGOLOG *programs as first-order terms. Then for every model* M *of* AX, $M \models transPr^*(\sigma, s, \delta, s') > 0$ *if and only if there exist* $\sigma_1, s_1, ..., \sigma_n, s_n$ *such that* $\sigma_1 = \sigma, s_1 = s, \sigma_n = \delta, s_n = s'$ *and* $M \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ *for* $i = 1, ..., n-1$.

Proof: (The proof is analogous to the proof of Lemma 2 in the Appendix of [dGLL00]).

$\Leftarrow$ By induction on n. If $n = 1$, then $M \models \mathit{transPr}^*(\sigma, s, \sigma, s) > 0$ by the definition of $\mathit{transPr}^*$. If $n > 1$, then by induction hypothesis $M \models \mathit{transPr}^*(\sigma_1, s_1, \sigma_{n-1}, s_{n-1}) > 0$. Furthermore, since $M \models \mathit{transPr}(\sigma_{n-1}, s_{n-1}, \sigma_n, s_n)$ we get $M \models \mathit{transPr}^*(\sigma_1, s_1, \sigma_n, s_n) > 0$ by the definition of $\mathit{transPr}^*$.

$\Rightarrow$ Let $\mathcal{R}$ be the relation formed by the tuples $(\sigma_1, s_1, \sigma_n, s_n)$ such that there exist σ_1, s_1, $..., \sigma_n, s_n$ and $M \models \mathit{transPr}(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ for $i = 1, ...n-1$. It is easy to verify that (i) for all δ, s, $(\delta, s, \delta, s) \in \mathcal{R}$; (ii) for all $\delta, s, \delta', s', \delta'', s''$, $(\delta, s, \delta', s') \in \mathcal{R}$ and $M \models \mathit{transPr}(\delta', s', \delta'', s'') > 0$ implies $(\delta, s, \delta'', s'') \in \mathcal{R}$. □

Finally, based on $\mathit{transPr}^*$ we can define the probability that the execution of σ in s results in an execution trace s', that is the probability to end up in a final configuration with situation component s'. This is captured by the function $\mathit{doPr}(\sigma, s, s')$, which is defined as follows:

$$\begin{array}{l}\mathit{doPr}(\sigma, s, s') = \\ \quad \textbf{if } \exists\delta.\mathit{transPr}(\sigma, s, \delta, s') > 0 \wedge \mathit{Final}(\delta, s') \textbf{ then } \mathit{transPr}(\sigma, s, \delta, s') \\ \quad \textbf{else } 0.\end{array}$$

This definition is non-ambiguous because in every final configuration $\langle \delta, s' \rangle$ reachable by a sequence of transition starting in $\langle \sigma, s \rangle$ the situation s' uniquely determines its associated remaining program δ. We will prove this (Proposition 16) and other properties of pGOLOG in Section 6.1.4.

6.1.2 An Example

In order to illustrate the semantics of pGOLOG, we will now discuss the different ways the world may evolve when the program *paintProc* is executed, assuming that initially the widget is neither processed nor painted.

Let AX be the foundational axioms of the epistemic situation calculus together with the definitions of *transPr*, *Final*, $\mathit{transPr}^*$ and *doPr* of this section, the axioms needed for the encoding of pGOLOG programs as first-order terms, the precondition axiom for *waitFor* and the successor state axiom for *start*. Furthermore, let Γ be AX together with the fact $\neg PR(S_0) \wedge \neg PA(S_0)$ and appropriate successor state axiom for *FL*, *BL*, *PR*, *ER* and *PA* (that is, axioms of the form $\mathit{Poss}(a, s) \supset [PA(do(a, s)) \equiv a = setPA \vee a \neq clipPA \wedge PA(s)]$, and similarly for the other fluents). Then, from Γ we can deduce that *paintProc* can result in two possible execution traces. Intuitively, the two possible execution traces correspond to the possible results of the prob instruction in the definition of *paintProc*. If the first branch is taken, this results in the execution of *setPA*, which causes *PA* to become true. If the second branch is taken, the execution trace simply ends with *tossTail*.

$$\begin{array}{l}\Gamma \models \mathit{doPr}(paintProc, S_0, s') = p \wedge p > 0 \equiv \\ \quad [s' = do([clipBL, tossHead, setPA], S_0) \wedge PA(s') \wedge p = 0.95 \vee \\ \quad\ s' = do([clipBL, tossTail], S_0) \wedge \neg PA(s') \wedge p = 0.05]\end{array}$$

Proof: Similar to earlier proofs dealing with the execution of a ConGolog or cc-Golog program, this proof is straightforward but laborious. By Proposition 8, we can prove the thesis by showing that there is a sequence of transitions from $\langle paintProc, S_0 \rangle$ to a final configuration with situation component $do([clipBL, tossHead, setPA], S_0)$ with total weight 0.95, and another sequence of transitions to a final configuration with situation component

$do([clipBL, tossTail], S_0)$. Furthermore, we have to show that these are the only sequences that lead to a final configuration.

First, from the definition of *transPr* and the fact $\neg PR(S_0)$, we get:

$$\begin{aligned} &transPr(\mathsf{if}(PR, setER, [clipBL, \mathsf{prob}(0.95, setPA), \mathsf{nil}]), S_0, \delta, s') = p \equiv \\ &\quad transPr([clipBL, \mathsf{prob}(0.95, setPA), \mathsf{nil}], S_0, \delta, s') = p. \end{aligned} \tag{6.3}$$

From the fact that *clipBL* is always possible:

$$\begin{aligned} &transPr([clipBL, \mathsf{prob}(0.95, setPA, \mathsf{nil})], S_0, \delta, s') = p \equiv \\ &\quad [p = 1 \wedge s' = do(clipBL, S_0) \wedge \delta = [\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})] \vee \\ &\quad p = 0 \wedge \neg(s' = do(clipBL, S_0) \wedge \delta = [\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})])]. \end{aligned} \tag{6.4}$$

Thus, there is only one successor configuration with positive weight. From the definition of *transPr* regarding prob, we get:

$$\begin{aligned} &transPr([\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})], do(clipBL, S_0), \delta, s') = p \equiv \\ &\quad [p = 0.95 \wedge s' = do([clipBL, tossHead], S_0) \wedge \delta = setPA \vee \\ &\quad p = 0.05 \wedge s' = do([clipBL, tossTail], S_0) \wedge \delta = \mathsf{nil} \vee \\ &\quad p = 0 \wedge \neg[s' = do([clipBL, tossHead], S_0) \wedge \delta = setPA \vee \\ &\qquad\qquad s' = do([clipBL, tossTail], S_0) \wedge \delta = \mathsf{nil}]]. \end{aligned} \tag{6.5}$$

Let us first consider the first branch. Here, from the fact that *setPA* is always possible:

$$\begin{aligned} &transPr(setPA, do([tossHead, clipBL], S_0), \delta, s') = p \equiv \\ &\quad [p = 1 \wedge s' = do([clipBL, tossHead, setPA], S_0) \wedge \delta = \mathsf{nil} \vee \\ &\quad p = 0 \wedge \neg(s' = do([clipBL, tossHead, setPA], S_0) \wedge \delta = \mathsf{nil})]. \end{aligned} \tag{6.6}$$

The new configuration has nil as program component, and is thus final. That is, there is a sequence of transitions from $\langle paintProc, S_0\rangle$ to $\langle \mathsf{nil}, do([clipBL, tossHead, setPA], S_0)\rangle$. From 6.3, 6.4, 6.5 and 6.6 we get that the product of the weights along this sequence is 1*0.95*1 = 0.95.

Next, let us consider the second disjunct in 6.5, that is the other configuration with positive weight, $\langle \mathsf{nil}, do([clipBL, tossHead], S_0)\rangle$. It is easy to see that this configuration is *Final*, meaning that we have found another sequence of transitions from $\langle paintProc, S_0\rangle$ to a final configuration. Furthermore, this configuration has $do([clipBL, tossHead], S_0)$ as situation component. From 6.3, 6.4 and 6.5 we get that the product of the weights along this sequence is 1*0.05 = 0.05.

Finally, from 6.3, 6.4, 6.5 and 6.6 we get that there are no other sequences of transition from $\langle paintProc, S_0\rangle$ to a final configuration, which finishes the proof. □

6.1.3 Extending the Semantics to Procedures

Let us now extend the semantics of pGOLOG to procedures. Similar to the case of ConGolog and cc-Golog described in the Sections 3.2.3 and 4.2.1, this is done by providing a second-order definition of *transPr* (the definitions of $transPr^*$ and *doPr* remain unaffected). In particular, we have to consider the following three additional program constructs:

- Procedure definitions $\{\mathcal{E}; \sigma\}$, where $\mathcal{E}$ is an environment and σ a program extended with procedure calls;

- Procedure calls $P(\vec{t})$, where P is a procedure name and $\vec{t}$ actual parameters associated to the procedure P;
- Contextualized procedure calls $[\mathcal{E} : P(\vec{t})]$, where $\mathcal{E}$ is an environment, P a procedure name and $\vec{t}$ actual parameters associated to the procedure P.

Because we want to show some important properties of the second-order versions of *Final* and *transPr*, we define them in a syntactically somewhat different form than that used to define *Final* and *Trans* in ConGolog and cc-Golog. In particular, we do not define *transPr* as the smallest set that satisfies a set of equivalence assertions ($\Phi \equiv \Psi$), but instead as the smallest set that satisfies a set of implications ($\Phi \supset \Psi$). Similarly for *Final*. The implication form turned out to be more convenient because it allows us to prove properties simply by showing that the property is closed under the implications that inductively define *transPr* respectively *Final*.[3]

First, we reformulate the second-order definition of *Final*:

$$Final(\sigma, s) \equiv \forall F.[... \supset F(\sigma, s)],$$

where the ellipsis stands for the universal closure of the conjunction of the following set of implications:

$$\begin{aligned}
\textsc{True} &\supset F(\mathsf{nil}, s)\\
F(\sigma_1, s) \wedge F(\sigma_2, s) &\supset F([\sigma_1, \sigma_2], s)\\
\phi[s] \wedge F(\sigma_1, s) &\supset F(\mathsf{if}(\phi, \sigma_1, \sigma_2), s)\\
\neg\phi[s] \wedge F(\sigma_2, s) &\supset F(\mathsf{if}(\phi, \sigma_1, \sigma_2), s)\\
\neg\phi[s] &\supset F(\mathsf{while}(\phi, \sigma), s)\\
F(\sigma, s) &\supset F(\mathsf{while}(\phi, \sigma), s)\\
F(\sigma, s) \wedge \phi[s] &\supset F(\mathsf{withCtrl}(\phi, \sigma), s)\\
F(\sigma_1, s) &\supset F(\mathsf{conc}(\sigma_1, \sigma_2), s)\\
F(\sigma_2, s) &\supset F(\mathsf{conc}(\sigma_1, \sigma_2), s)\\
F(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s) &\supset F(\{\mathcal{E}; \sigma\}, s)\\
F(\{\mathcal{E}; \beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\}, s) &\supset F([\mathcal{E} : P(\vec{t})], s).
\end{aligned}$$

Proposition 9: *The above definition for Final is equivalent to cc-Golog's definition of Final.*

Proof: Similar to [dGLL00] (their proof of Theorem 10), we show the equivalence of the above definition and the definition for cc-Golog using the Tarski-Knaster fixpoint theorem [Tar55]. The Tarski-Knaster fixpoint theorem says that if

$$S(\vec{x}) \equiv \forall Z.[[\forall \vec{y}.\Phi(Z, \vec{y}) \supset Z(\vec{y})] \supset Z(\vec{x})] \tag{6.7}$$

[3]The idea to define *transPr* in terms of a set implications is inspired by [dGLL00], who reformulate a subset of their definition of *Trans* in implication form in their Appendix B.

and $\Phi(Z, \vec{y})$ is monotonic, then we get the following consequence:

$$S(\vec{x}) \equiv \forall Z.[[\forall \vec{y}.Z(\vec{y}) \equiv \Phi(Z, \vec{y})] \supset Z(\vec{x})]. \tag{6.8}$$

Here, $\Phi(Z, \vec{y})$ is called monotonic if and only if $\forall Z_1, Z_2.[\forall \vec{y}.Z_1(\vec{y}) \supset Z_2(\vec{y})] \supset [\forall \vec{y}.\Phi(Z_1, \vec{y}) \supset \Phi(Z_2, \vec{y})]$. A sufficient condition for monotonicity is that all occurrences of Z occur within an even number of negations (note that $\Phi \supset \Psi$ is an abbreviation for $\neg\Phi \vee \Psi$).

Now similar to the proof of Proposition 5 (Page 68), the above definition of *Final* can be put in the form (6.7). In particular, each of the implications in the above definition of *Final* can be rewritten so that only the variables s, σ appear in the right-hand part of the implications. The form (6.7) can then be obtained by getting the disjunction of all left-hand sides, which have the following form:

$$\begin{aligned}
\sigma &= \mathsf{nil} \\
F(\sigma_1, s) \wedge F(\sigma_2, s) \wedge \sigma &= [\sigma_1, \sigma_2] \\
\phi[s] \wedge F(\sigma_1, s) \wedge \sigma &= \mathsf{if}(\phi, \sigma_1, \sigma_2) \\
\neg\phi[s] \wedge F(\sigma_2, s) \wedge \sigma &= \mathsf{if}(\phi, \sigma_1, \sigma_2) \\
\neg\phi[s] \wedge \sigma &= \mathsf{while}(\phi, \gamma) \\
F(\gamma, s) \wedge \sigma &= \mathsf{while}(\phi, \gamma) \\
F(\gamma, s) \wedge \phi[s] \wedge \sigma &= \mathsf{withCtrl}(\phi, \gamma) \\
F(\sigma_1, s) \wedge \sigma &= \mathsf{conc}(\sigma_1, \sigma_2) \\
F(\sigma_2, s) \wedge \sigma &= \mathsf{conc}(\sigma_1, \sigma_2) \\
F(\gamma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s) \wedge \sigma &= \{\mathcal{E}; \gamma\} \\
F(\{\mathcal{E}; \beta_{P^{\vec{v_p}}_{\vec{t}[s]}}\}, s) \wedge \sigma &= [\mathcal{E} : P(\vec{t})].
\end{aligned}$$

Now it is easy to see that the resulting formula Φ is indeed monotonic because the predicate variable F does not occur in the scope of any negation. Thus, by the Tarski-Knaster fixpoint theorem, the above definition can be rewritten in the form (6.8). Once in this form it is easy to see that the above definition is equivalent to cc-Golog's definition of *Final*; cf. the proof of Proposition 5. □

As an aside, we remark that unfortunately, neither *Trans* nor *transPr* is syntactically monotonic, so we cannot use the Tarski-Knaster fixpoint theorem to convert the second-order definition of *transPr* to the form (6.8).

Similar to *Final*, the possible transitions for pGOLOG programs with procedures are specified by the following second-order definition of *transPr*:

$$\begin{aligned}
transPr(\sigma, s, \delta, s') = p \equiv &\forall t.[\Phi_{\text{transPr}} \supset t(\sigma, s, \delta, s') = p] \vee \\
p = 0 \wedge \neg\exists p'.&\forall t[\Phi_{\text{transPr}} \supset t(\sigma, s, \delta, s') = p']
\end{aligned} \tag{6.9}$$

where Φ_{transPr} stands for the universal closure of the conjunction of the following set of implications:

$$Poss(\alpha[s], s) \supset t(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1$$

$$\phi[s] \supset t(\phi?, s, \mathsf{nil}, s) = 1$$

$$t(\sigma_1, s, \gamma, s') = p \wedge p > 0 \supset t([\sigma_1, \sigma_2], s, [\gamma, \sigma_2], s') = p$$

$$Final(\sigma_1, s) \wedge t(\sigma_2, s, \delta, s') = p \wedge p > 0 \supset t([\sigma_1, \sigma_2], s, \delta, s') = p$$

$$\phi[s] \wedge t(\sigma_1, s, \delta, s') = p \wedge p > 0 \supset t(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') = p$$

$$\neg\phi[s] \wedge t(\sigma_2, s, \delta, s') = p \wedge p > 0 \supset t(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') = p$$

$$\phi[s] \wedge t(\sigma, s, \gamma, s') = p \wedge p > 0 \supset t(\mathsf{while}(\phi, \sigma), s, [\gamma, \mathsf{while}(\phi, \sigma)], s') = p$$

$$\phi[s] \wedge t(\sigma, s, \gamma, s') = p \wedge p > 0 \supset t(\mathsf{withCtrl}(\phi, \sigma), s, \mathsf{withCtrl}(\phi, \gamma), s') = p$$

$$\begin{aligned}
&\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge \\
&\quad t(\sigma_1, s, \delta_1, s') = p \wedge p > 0 \wedge \\
&\quad\quad [\forall \delta_2, s_2 . t(\sigma_2, s, \delta_2, s_2) > 0 \\
&\quad\quad\quad \supset start(s') \leq start(s_2)] \supset t(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s') = p
\end{aligned}$$

$$\begin{aligned}
&\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge \\
&\quad t(\sigma_2, s, \delta_2, s') = p \wedge p > 0 \wedge \\
&\quad\quad [\forall \delta_1, s_1 . t(\sigma_1, s, \delta_1, s_1) > 0 \\
&\quad\quad\quad \supset start(s') < start(s_1)] \supset t(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\sigma_1, \delta_2), s') = p
\end{aligned}$$

$$\textsc{True} \supset t(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_1, do(tossHead, s)) = p$$

$$\textsc{True} \supset t(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_2, do(tossTail, s)) = 1 - p$$

$$t(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s') = p \wedge p > 0 \supset t(\{\mathcal{E}; \sigma\}, s, \delta, s') = p$$

$$t(\{\mathcal{E}; \beta_{P^{\vec{v_p}}_{\vec{t[s]}}}\}, s, \delta, s') = p \wedge p > 0 \supset t([\mathcal{E} : P(\vec{t})], s, \delta, s') = p.$$

Let us now consider the definition of *transPr* in more detail, beginning with definition (6.9). It is somewhat more complex than its counterpart (3.43) on page 51 because it does not only specify which configurations $\langle \delta, s' \rangle$ can be reached but also specifies the probability to reach them. To understand how the definition works, let us first consider the first disjunct of (6.9). This disjunct deals with the case where Φ_{transPr} uniquely implies a weight p for the transition from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$. If this is the case, then p is also the weight of $transPr(\sigma, s, \delta, s')$. On the other hand, if Φ_{transPr} does not uniquely implies a weight p for the transition from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$ then $transPr(\sigma, s, \delta, s') = 0$.

Next, let us consider the implications in Φ_{transPr}. The first eleven implications (all implications except for the last two) take care of programs without procedures, and thus take the role of the first-order definition of *transPr*. Note that they only state – by means of implications – in which cases $t(\sigma, s, \delta, s')$ is required to have a positive value. For example, the first implication deals with the case of primitive actions, saying that if a primitive action α is possible in s, then $t(\alpha, s, \mathsf{nil}, do(\alpha[s], s))$ must have value 1. However, nothing is said about the case where α is not possible in s. This is unlike in the first-order definition of

transPr where we also explicitly stated when a transition has probability 0. Intuitively, in the implications of Φ_{transPr} it is not necessary to consider the cases where $t(\sigma, s, \delta, s')$ must have value 0 because by (6.9) *transPr* is defined to have value 0 whenever Φ_{transPr} does not require all functions t to have the same (positive) value.

Finally, let us consider the last two implications of the set Φ_{transPr}. As in Section 3.2.3, $\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}$ denotes the program σ with all procedures bound by $\mathcal{E}$ and free in σ replaced by their contextualized version, and $\beta_{P^{\vec{v_p}}_{\vec{t}[s]}}$ denotes the body of the procedure P in $\mathcal{E}$ with formal parameters $\vec{v_p}$ substituted by the actual parameters $\vec{t}$ evaluated in the current situation. Here, the usual notion of free and bound apply, so for example in $\{\mathsf{proc}(P_1(), a); [P_2, P_1]\}$, P_1 is bound but P_2 is free. The first of the two axioms says that to show that a program σ with an associated environment $\mathcal{E}$ can cause a transition, one has to show that the program which results from σ by simultaneously substituting all procedure calls bound by $\mathcal{E}$ with procedure calls contextualized by the environment of the procedure can cause a transition. The second axiom says that to show that a contextualized procedure call can cause a transition one has to show that the body of the procedure, with the formal parameters substituted with the actual parameters evaluated in the current situation, and associated with $\mathcal{E}$ in order to deal with further procedure calls, can cause a transition.

Similar to the case of cc-Golog, we have the following proposition:

Proposition 10: *With respect to pGOLOG programs without procedures, the first-order and the second-order definitions of transPr and Final are equivalent.*

Proof: The proof for *Final* is analogous to the proof for *transPr*, so we only consider the latter. Let us now denote *transPr* defined by the second-order sentence as $transPr_{SOL}$ and the earlier first-order definition of *transPr* as $transPr_{FOL}$. Since procedures are not considered, we can drop, without loss of generality, the assertions for contextualized procedures and procedures with an environment in the definition of $transPr_{SOL}$. Then:

- $transPr_{SOL}(\sigma, s, \delta, s') = p \wedge p > 0 \supset transPr_{FOL}(\sigma, s, \delta, s') = p$.

  By definition 6.9, $transPr_{SOL}(\sigma, s, \delta, s') = p \wedge p > 0$ implies $\forall t.[\Phi_{\text{transPr}} \supset t(\sigma, s, \delta, s') = p]$. Thus, to prove $transPr_{FOL}(\sigma, s, \delta, s') = p$ it suffices to show that $transPr_{FOL}$ satisfies the set of implications Φ_{transPr} (with $transPr_{FOL}$ as instance of the variable t). We only give details for a few cases; the other cases are analogous.

  - Primitive action. We have to show $Poss(\alpha[s], s) \supset transPr_{FOL}(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1$. This holds by the definition of $transPr_{FOL}$.
  - First implication for sequences. We have to show

    $$transPr_{FOL}(\sigma_1, s, \gamma, s') = p \wedge p > 0 \supset transPr_{FOL}([\sigma_1, \sigma_2], s, [\gamma, \sigma_2], s') = p.$$

    This holds by the definition of $transPr_{FOL}$.
  - Second implication for sequences. As above,

    $$Final(\sigma_1, s) \wedge transPr_{FOL}(\sigma_2, s, \delta, s') = p \wedge p > 0 \supset transPr_{FOL}([\sigma_1, \sigma_2], s, \delta, s') = p$$

    holds by the definition of $transPr_{FOL}$.
  - First implication for concurrent execution. We have to show

$[\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge transPr_{FOL}(\sigma_1, s, \delta_1, s') = p \wedge p > 0 \wedge$
$\forall \delta_2, s_2. transPr_{FOL}(\sigma_2, s, \delta_2, s_2) > 0 \supset start(s') \leq start(s_2)]$
$\supset transPr_{FOL}(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s') = p$

This holds by the definition of $transPr_{FOL}$.

- $transPr_{FOL}(\sigma, s, \delta, s') = p \wedge p > 0 \supset transPr_{SOL}(\sigma, s, \delta, s') = p$.

  By induction on the structure of σ considering as base cases nil, a, and $\phi?$, and then applying the induction argument. We only give details for primitive actions and sequences. The other cases are analogous.

  – Base case: primitive action.

  By the definition of $transPr_{FOL}$, $transPr_{FOL}(\alpha, s, \delta, s') = p \wedge p > 0$ if and only if $Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s)$, in which case p is 1. Thus, we have to show that $Poss(\alpha[s], s)$ implies $transPr_{SOL}(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1$.

  By definition 6.9, $transPr_{SOL}(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = p, p > 0$ if and only if any function t satisfying Φ_{transPr} fulfills $t(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = p$. Consider any function t satisfying Φ_{transPr}. By the first implication of Φ_{transPr}, $t(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1$.

  – Induction step: sequence.

  By the definition of $transPr_{FOL}$, $transPr_{FOL}([\sigma_1, \sigma_2], s, \delta, s') = p \wedge p > 0$ if and only if either (a) $\exists \gamma. \delta = [\gamma, \sigma_2] \wedge transPr_{FOL}(\sigma_1, s, \gamma, s') = p$ or (b) $Final(\sigma_1, s) \wedge transPr_{FOL}(\sigma_2, s, \delta, s') = p$.

  Let us first consider case (a). By induction hypothesis, $transPr_{SOL}(\sigma_1, s, \gamma, s') = p$. Thus, by definition 6.9 any function t satisfying Φ_{transPr} fulfills $t(\sigma_1, s, \gamma, s') = p$. Then, by Φ_{transPr}'s first implication for sequences we get $t([\sigma_1, \sigma_2], s, \delta, s') = p$ for any function t satisfying Φ_{transPr}, and hence $transPr_{SOL}([\sigma_1, \sigma_2], s, \delta, s') = p$. Case (b) can be shown analogously, using Φ_{transPr}'s second implication for sequences.

□

To illustrate the new second-order definition of *transPr*, let us now consider the example program $\{\mathcal{E}_1; paintProc()\}$, where the procedure *paintProc* is defined by $\mathcal{E}_1$ as follows:

$\mathcal{E}_1 \doteq \mathsf{proc}(paintProc, \mathsf{if}(PR, setER, [clipBL, \mathsf{prob}(0.95, setPA)]))$.

Then, by the definition of Φ_{transPr} we get:

$t(\mathsf{if}(PR, setER, [clipBL, \mathsf{prob}(0.95, setPA)]), s, \delta, s') = p \wedge p > 0 \supset$
$t(\{\mathcal{E}_1; \mathsf{if}(PR, setER, [clipBL, \mathsf{prob}(0.95, setPA)])\}, s, \delta, s') = p \wedge p > 0 \supset$
$t([\mathcal{E}_1 : paintProc()], s, \delta, s') = p \wedge p > 0 \supset$
$t(\{\mathcal{E}_1; paintProc()\}, s, \delta, s') = p.$

Thus, if in s the body of *paintProc* can cause a transition with positive weight p to a successor configuration $\langle \delta, s' \rangle$, then $\{\mathcal{E}_1; paintProc()\}$ can cause the same transition with weight p. In particular, from Proposition 10 and the considerations from Section 6.1.2 where we considered an execution of the body of *paintProc* we can conclude

$$transPr(\{\mathcal{E}_1; paintProc()\}, S_0, [\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})], do(clipBL, S_0)) = 1.$$

To see that this is the only possible transition (that is the only transition with positive probability) of $\{\mathcal{E}_1; paintProc()\}$ in S_0 to a successor configuration, observe that Φ_{transPr} does not imply any other positive implications for the value of $t(\{\mathcal{E}_1; paintProc()\}, S_0, \delta, s')$. Thus one can always choose a function t_0 that satisfies Φ_{transPr} such that

$$t_0(\{\mathcal{E}_1; paintProc()\}, S_0, \delta, s') = 0 \equiv \\ \neg(s' = do(clipBL, S_0) \wedge \delta = [\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})]).$$

By the definition of *transPr* (6.9), then, we get:

$$transPr(\{\mathcal{E}_1; paintProc()\}, S_0, \delta, s') = 0 \equiv \\ \neg(s' = do(clipBL, S_0) \wedge \delta = [\mathsf{nil}, \mathsf{prob}(0.95, setPA, \mathsf{nil})]).$$

Similarly, we can conclude that the overall execution of $\{\mathcal{E}_1; paintProc()\}$ in S_0 can result in exactly the following two execution traces:

- $do([clipBL, tossHead, setPA], S_0)$, and
- $do([clipBL, tossTail], S_0)$.

Finally, to see that the new definition of *transPr* correctly deals with circular recursive procedure, let us consider the ill-formed program $\{\mathsf{proc}(P(), P()); P()\}$. Let $\mathcal{E}_2$ be an abbreviation for $\mathsf{proc}(P(), P())$. The only implications we get by Φ_{transPr} about the value of $t(\{\mathcal{E}_2; P\}, s, \delta, s')$ are:

$$t(\{\mathcal{E}_2; P\}, s, \delta, s') = p \supset \\ t([\mathcal{E}_2 : P], s, \delta, s') = p \supset \\ t(\{\mathcal{E}_2; P\}, s, \delta, s') = p.$$

Thus, we can choose two functions t_0 and t_1 which both satisfy the implication in Φ_{transPr} and where $t_0(\{\mathcal{E}_2; P\}, s, \delta, s') = 0$ and $t_1(\{\mathcal{E}_2; P\}, s, \delta, s') = 1$. So there is no p' such that $\forall t[\Phi_{\text{transPr}} \supset t(\{\mathcal{E}_2; P\}, s, \delta, s') = p']$, and hence $transPr(\{\mathcal{E}_2; P\}, s, \delta, s') = 0$. We remark that $\{\{\mathcal{E}_2; P\}, s, \delta, s')\}$ is not final, meaning that the ill-formed program cannot perform a transition with positive probability and at the same time is not final.

6.1.4 Formal Properties

We will now investigate some important properties of *transPr*, *Final* and *doPr*. In particular, we show that *doPr* is well defined because in every final configuration $\langle \delta, s' \rangle$ reachable by a sequence of transitions starting in $\langle \sigma, s \rangle$ the situation s' uniquely determines its associated remaining program δ (Proposition 16). The following properties will also be helpful in the next chapter, where we specify how the robot is to update its probabilistic beliefs. We remark that Proposition 8 still holds for the new second-order definition of *transPr* (recall that the proof does not rely on any particular features of *transPr*). To simplify the presentation of the proofs, we will use the same symbols to denote terms and elements of the domain of interpretation; the meaning will be clear from the context.

The first proposition tells us that to show that the tuples σ, s, δ, s' such that there is transition from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$ have a property P, it suffices to prove that P satisfies the

following set of implications Φ_{Prop}:

$$\begin{aligned}
Poss(\alpha[s], s) &\supset P(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) \\
\phi[s] &\supset P(\phi?, s, \mathsf{nil}, s) \\
P(\sigma_1, s, \gamma, s') &\supset P([\sigma_1, \sigma_2], s, [\gamma, \sigma_2], s') \\
Final(\sigma_1, s) \wedge P(\sigma_2, s, \delta, s') &\supset P([\sigma_1, \sigma_2], s, \delta, s') \\
\phi[s] \wedge P(\sigma_1, s, \delta, s') &\supset P(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') \\
\neg\phi[s] \wedge P(\sigma_2, s, \delta, s') &\supset P(\mathsf{if}(\phi, \sigma_1, \sigma_2), s, \delta, s') \\
\phi[s] \wedge P(\sigma, s, \gamma, s') &\supset P(\mathsf{while}(\phi, \sigma), s, [\gamma, \mathsf{while}(\phi, \sigma)], s') \\
\phi[s] \wedge P(\sigma, s, \gamma, s') &\supset P(\mathsf{withCtrl}(\phi, \sigma), s, \mathsf{withCtrl}(\phi, \gamma), s') \\
\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge P(\sigma_1, s, \delta_1, s') \wedge & \\
[\forall \delta_2, s_2. P(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)] &\supset P(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s') \\
\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge P(\sigma_2, s, \delta_2, s') \wedge & \\
[\forall \delta_1, s_1. P(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1)] &\supset P(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\sigma_1, \delta_2), s') \\
\textsc{True} &\supset P(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_1, do(tossHead, s)) \\
\textsc{True} &\supset P(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_2, do(tossTail, s)) \\
P(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s') &\supset P(\{\mathcal{E}; \sigma\}, s, \delta, s') \\
P(\{\mathcal{E}; \beta_P{}^{\vec{v_p}}_{\vec{t[s]}}\}, s, \delta, s') &\supset P([\mathcal{E} : P(\vec{t})], s, \delta, s')
\end{aligned}$$

Note that the above set of implications corresponds, roughly, to Φ_{transPr} with the condition $t(\sigma, s, \delta, s') > 0$ replaced by $P(\sigma, s, \delta, s')$.

Proposition 11: *Let* $\text{AX}_{\text{transPr}}$ *be the foundational axioms of the epistemic situation calculus together with the precondition axiom for waitFor, the successor state axiom for start, the (second-order) definitions of transPr, Final and transPr* plus the axioms needed for the encoding of pGOLOG programs as first-order terms. Then:*

$$\text{AX}_{\text{transPr}} \models transPr(\sigma, s, \delta, s') > 0 \supset \forall P.[\Phi_{Prop} \supset P(\sigma, s, \delta, s')].$$

Proof: First, note that $transPr(\sigma, s, \delta, s') > 0$ is equivalent to $\forall t.[\Phi_{\text{transPr}} \supset t(\sigma, s, \delta, s') > 0]$. The latter is equivalent to $\forall t.[\Phi^1_{\text{transPr}} \supset t(\sigma, s, \delta, s') > 0]$, where Φ^1_{transPr} is obtained from Φ_{transPr} by replacing the two implication for prob instructions by

$$\begin{aligned}
\textsc{True} &\supset t(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_1, do(tossHead, s)) = 1 \\
\textsc{True} &\supset t(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_2, do(tossTail, s)) = 1.
\end{aligned}$$

In particular, for every function t which satisfies Φ_{transPr} there is exactly one function t^1 which satisfies Φ^1_{transPr} and which has positive value at σ, s, δ, s' if and only if t has positive value at σ, s, δ, s'. Note that we have restricted prob instructions to a value p which satisfies $0 <$

$p < 1$, hence both p and $1-p$ are > 0. Finally, note that the different implications in Φ_{transPr} are mutually exclusive since each of them deals with programs of a specific form, and we have required unique names for programs and situations (cf. Section 3.1.1 and Appendix A).

Next, we show that for all σ, s, δ, s', $\forall t.[\Phi^1_{\text{transPr}} \supset t(\sigma, s, \delta, s') > 0]$ implies $\forall P.[\Phi_{\text{Prop}} \supset P(\sigma, s, \delta, s')]$ by proving that for all σ, s, δ, s', $\exists P.\Phi_{\text{Prop}} \wedge \neg P(\sigma, s, \delta, s')$ implies $\exists t.\Phi^1_{\text{transPr}} \wedge \neg t(\sigma, s, \delta, s') > 0$. Let $\sigma_0, s_0, \delta_0, s'_0$ be an arbitrary tuple, and suppose that there is a relation Q which satisfies Φ_{Prop} such that $\neg Q(\sigma_0, s_0, \delta_0, s'_0)$ holds. We will use this relation to construct a function t_Q which satisfies Φ^1_{transPr} but assigns value 0 to $\sigma_0, s_0, \delta_0, s'_0$. In particular, t_Q is defined as follows:

$$t_Q(\sigma, s, \delta, s') = p \equiv Q(\sigma, s, \delta, s') \wedge p = 1 \vee \neg Q(\sigma, s, \delta, s') \wedge p = 0. \tag{6.10}$$

That is, $t_Q(\sigma, s, \delta, s')$ has value 1 wherever $Q(\sigma, s, \delta, s')$ holds and else has value 0. In particular, $t_Q(\sigma_0, s_0, \delta_0, s'_0) = 0$. Next, we show that the function t_Q satisfies Φ^1_{transPr} by verifying that t_Q satisfies all the implications in Φ^1_{transPr}. We only give details for the most important cases:

- Primitive action. If $\alpha[s]$ is possible in s, then $Q(\alpha, s, \mathsf{nil}, do(\alpha[s], s))$ holds because Q satisfies (the first implication of) Φ_{Prop}. Thus, $t_Q(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1$.

- Second implication for sequences. We have to show

  $$Final(\sigma_1, s) \wedge t_Q(\sigma_2, s, \delta, s') = p \wedge p > 0 \supset t_Q([\sigma_1, \sigma_2], s, \delta, s') = p.$$

  Assume t_Q satisfies $t_Q(\sigma_2, s, \delta, s') = p \wedge p > 0$ (else, there is nothing to be shown). Then, by construction of t_Q (6.10) p must be 1 and $Q(\sigma_2, s, \delta, s')$ must hold. Hence, by Φ_{Prop} (second implication for sequences) we get $Q([\sigma_1, \sigma_2], s, \delta, s')$, and thus by construction of t_Q $t_Q([\sigma_1, \sigma_2], s, \delta, s') = 1 = p$.

- First implication for concurrent execution. We have to show

  $$\begin{aligned}&\neg Final(\sigma_1, s) \wedge \neg Final(\sigma_2, s) \wedge t_Q(\sigma_1, s, \delta_1, s') = p \wedge p > 0 \wedge \\ &\quad [\forall \delta_2, s_2.t_Q(\sigma_2, s, \delta_2, s_2) > 0 \supset start(s') \leq start(s_2)] \\ &\qquad \supset t_Q(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s') = p\end{aligned}$$

  Assume t_Q satisfies the left-hand-side of the implication (else, there is nothing to be shown). Then, by the construction of t_Q (6.10) we get:

  $$\begin{gathered}t_Q(\sigma_1, s, \delta_1, s') = 1 \\ Q(\sigma_1, s, \delta_1, s') \\ [\forall \delta_2, s_2.Q(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)].\end{gathered}$$

  Now from this and Φ_{Prop} (second implication for concurrent execution) it follows that $Q(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s')$ holds, and hence $t_Q(\mathsf{conc}(\sigma_1, \sigma_2), s, \mathsf{conc}(\delta_1, \sigma_2), s') = 1$.

- First implication for prob. $Q(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_1, do(tossHead, s))$ holds because Q satisfies Φ_{Prop} (first implication for prob). Thus, $t_Q(\mathsf{prob}(p, \sigma_1, \sigma_2), s, \sigma_1, do(tossHead, s)) = 1$.

- Program with associated environment. We have to show

$$t_Q(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s') = p \wedge p > 0 \supset t_Q(\{\mathcal{E};\sigma\}, s, \delta, s') = p.$$

Assume t_Q satisfies the left-hand-side of the implication. Then by the construction of t_Q we get $t_Q(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s') = 1$ and $Q(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s, \delta, s')$. Furthermore, by Φ_{Prop}, $Q(\{\mathcal{E};\sigma\}, s, \delta, s')$ holds. Thus, $t_Q(\{\mathcal{E};\sigma\}, s, \delta, s') = 1$.

□

The next proposition says that there is no configuration $\langle\sigma, s\rangle$ which at the same time is final and has a successor configuration with positive weight.

Proposition 12: *Let* $\text{AX}_{\text{transPr}}$ *be as above. Then:*

$$\text{AX}_{\text{transPr}} \models \mathit{transPr}(\sigma, s, \delta, s') > 0 \supset \neg\mathit{Final}(\sigma, s).$$

Proof: By Proposition 11, it suffices to show that *Final* satisfies the set of implications Φ_{Prop}. We use the fact that by Proposition 9 the set of axioms in the definition of *Final* can be rewritten in the form $\Phi \equiv \Psi$ (see page 67).

- Primitive action. We have to show $\mathit{Poss}(\alpha[s], s) \supset \neg\mathit{Final}(\alpha, s)$. This holds by the definition of *Final* because a primitive action is never final.
- Test. A test is never final
- First implication for sequences. We have to show $\neg\mathit{Final}(\sigma_1, s) \supset \neg\mathit{Final}([\sigma_1, \sigma_2], s)$. This holds by definition of *Final*.
- Second implication for sequences. We have to show $\mathit{Final}(\sigma_1, s) \wedge \neg\mathit{Final}(\sigma_2, s) \supset \neg\mathit{Final}([\sigma_1, \sigma_2], s)$, which holds by the definition of *Final*.
- First implication for conditionals. We have to show that $\phi[s] \wedge \neg\mathit{Final}(\sigma_1, s)$ implies $\neg\mathit{Final}(\mathsf{if}(\phi, \sigma_1, \sigma_2), s)$, which holds by the definition of *Final*.
- Second implication for conditionals. We have to show $\neg\phi[s] \wedge \neg\mathit{Final}(\sigma_2, s, \delta, s') \supset \neg\mathit{Final}(\mathsf{if}(\phi, \sigma_1, \sigma_2))$, which holds by the definition of *Final*.
- While loops. We have to show $\phi[s] \wedge \neg\mathit{Final}(\sigma, s) \supset \neg\mathit{Final}(\mathsf{while}(\phi, \sigma), s)$, which holds by the definition of *Final*.
- First implication for concurrent execution. It suffices to show

  $$\neg\mathit{Final}(\sigma_1, s) \wedge \neg\mathit{Final}(\sigma_2, s) \supset \neg\mathit{Final}(\mathsf{conc}(\sigma_1, \sigma_2), s),$$

  which holds by the definition of *Final*. Analogous for the second implication for concurrent execution.
- First implication for prob. We have to show $\text{TRUE} \supset \neg\mathit{Final}(\mathsf{prob}(p, \sigma_1, \sigma_2))$, which holds by the definition of *Final* because prob is never *Final*. Analogous for the second implication for prob.

- Program with associated environment. We have to show

$$\neg Final(\sigma^{P_i(\vec{t})}_{[\mathcal{E}:P_i(\vec{t})]}, s) \supset \neg Final(\{\mathcal{E};\sigma\}, s),$$

  which holds by the definition of *Final*.

- Contextualized procedure call. We have to show

$$\neg Final(\{\mathcal{E}; \beta_{P^{\vec{v_p}}_{\vec{t}[\vec{s}]}}\}, s) \supset \neg Final([\mathcal{E} : P(\vec{t})], s),$$

  which holds by the definition of *Final*.

□

While the previous proposition says that a configuration cannot at the same time be final and have a successor configuration with positive weight, the following proposition says that if $\langle \delta, s' \rangle$ is a successor configuration of $\langle \sigma, s \rangle$ then s' can be obtained from s by executing at most one action. Furthermore, it says that a configuration can have at most two successor configurations with positive weight, and that if a configuration has two successor configurations, then these two configurations have syntactically different situation components.

Proposition 13: *Let* $\mathrm{AX}_{\mathrm{transPr}}$ *be as above. Then:*

- $\mathrm{AX}_{\mathrm{transPr}} \models transPr(\sigma, s, \sigma', s') > 0 \supset s = s' \vee \exists a. s' = do(a, s)$*; and*
- $\mathrm{AX}_{\mathrm{transPr}} \models transPr(\sigma, s, \sigma', s') > 0 \wedge transPr(\sigma, s, \sigma'', s'') > 0 \supset$
  $s' = s'' \wedge \sigma' = \sigma'' \vee$
  $s' = do(tossHead, s) \wedge s'' = do(tossTail, s) \vee$
  $s' = do(tossTail, s) \wedge s'' = do(tossHead, s).$

Proof: The first part of the proposition follows analogously to Proposition 12, using Proposition 11. Unfortunately, the second part of the proposition involves *two* occurrences of *transPr*, so we cannot prove it similarly.

To see that the second part holds, first note that Φ_{transPr}'s implications for sequences, conditionals, while-loops, withCtrl-constructs, concurrent execution, programs with associated environment and contextualized procedure calls are mutually exclusive in the sense that for every possible configuration $\langle \sigma, s \rangle$ there is at most one implication which can be used to derive that there is a successor configuration $\langle \sigma', s' \rangle$ such that $transPr(\sigma, s, \sigma', s') > 0$. In particular, the implications for syntactically different programs are mutually exclusive because programs have unique names. On the other hand, the two implications for sequences are mutually exclusive because, by Proposition 12, $transPr(\sigma_1, s, \gamma, s')$ implies $\neg Final(\sigma_1, s)$. Similarly, the two implications for concurrent execution are mutually exclusive because their antecedents contradict each other (we remark that this does not hold for ConGolog's $(\sigma_1 \parallel \sigma_2)$ construct, which is the reason why we do not consider unprioritized concurrency in pGOLOG). Finally, the antecedents of the two implications for conditional execution also contradict each other.

Next, observe that the implications for primitive actions, tests and prob instructions can be rewritten in the form

$$\Psi_i(\sigma, s, \sigma', s') \supset t(\sigma, s, \sigma', s') > 0 \tag{6.11}$$

where Ψ_i is a formula whose free variables are among σ, s, σ', s'. For example, the implication for primitive actions can be rewritten as

$$\sigma = \alpha[s] \wedge Poss(\sigma, s) \wedge \sigma' = \mathsf{nil} \wedge s' = do(\alpha[s], s) \supset t(\sigma, s, \sigma', s') = 1.$$

Similarly the implications for sequences, conditionals, while-loops, withCtrl-constructs, concurrent execution, programs with associated environment and contextualized procedure calls can all be rewritten in the following form:

$$t(\sigma_p, s, \sigma_p', s') > 0 \wedge \Psi_i(\sigma_p, \sigma, s, \sigma_p', \sigma', s') \supset t(\sigma, s, \sigma', s') > 0. \tag{6.12}$$

For example, the first implication for concurrent execution can be rewritten as follows:

$$\begin{array}{l} t(\sigma_p, s, \delta_p, s') = p \wedge p > 0 \wedge \\ \quad [\exists \sigma_2 . \sigma = \mathsf{conc}(\sigma_p, \sigma_2) \wedge \sigma' = \mathsf{conc}(\delta_p, \sigma_2) \wedge \neg Final(\sigma_p, s) \wedge \neg Final(\sigma_2, s) \wedge \\ \quad [\forall \delta_2, s_2 . t(\sigma_2, s, \delta_2, s_2) > 0 \supset start(s') \leq start(s_2)]] \\ \qquad \supset t(\sigma, s, \sigma', s') > 0 \end{array}$$

Furthermore, it is easy to verify that each such condition Ψ_i satisfies the following assertions:

$$\Psi_i(\sigma_p, \sigma, s, \sigma_p', \sigma', s') \wedge \Psi_i(\delta_p, \delta, s, \delta_p', \delta', s'') \wedge \sigma' \neq \delta' \wedge \sigma = \delta \supset \sigma_p = \delta_p \wedge \sigma_p' \neq \delta_p'; \tag{6.13}$$

$$\Psi_i(\sigma_p, \sigma, s, \sigma_p', \sigma', s') \wedge \Psi_i(\delta_p, \delta, s, \delta_p', \delta', s'') \wedge s' \neq s'' \wedge \sigma = \delta \supset \sigma_p = \delta_p. \tag{6.14}$$

The first assertion says that an implication of the form (6.12) only implies that a configuration $\langle \sigma, s \rangle$ has two successor configurations $\langle \sigma', s' \rangle$ and $\langle \delta', s'' \rangle$ with different program component if there is another configuration $\langle \sigma_p, s \rangle$ which has two successor configurations $\langle \sigma_p', s' \rangle$ and $\langle \delta_p', s'' \rangle$ with different program component. Similarly, the second assertion says that an implication of the form (6.12) only implies that a configuration $\langle \sigma, s \rangle$ has two successor configurations $\langle \sigma', s' \rangle$ and $\langle \delta', s'' \rangle$ with different situation component if there is another configuration $\langle \sigma_p, s \rangle$ which has two successor configurations $\langle \sigma_p', s' \rangle$ and $\langle \delta_p', s'' \rangle$ (which obviously have different situation component). Intuitively, this tells us that the implications for sequences, conditionals, while-loops, withCtrl-constructs, concurrent execution, programs with associated environment and contextualized procedure calls are not the ultimate "cause" of configurations with more than one successor configuration.

Next, observe that, by the definition of *transPr*, $transPr(\sigma, s, \sigma', s') > 0$ implies that all functions t satisfying Φ_{transPr} fulfill $t(\sigma, s, \sigma', s') > 0$. That is, $transPr(\sigma, s, \sigma', s') > 0$ implies that Φ_{transPr} *entails* $t(\sigma, s, \sigma', s') > 0$, where t is considered as an ordinary *first-order* function symbol. Due to the fact that Φ_{transPr} is equivalent to a set of implications of the form (6.12) and (6.11), this means that there must be a finite sequence of implications

$$\begin{array}{l} \Psi_1(\sigma_1, s, \sigma_1', s') \supset t(\sigma_1, s, \sigma_1', s') > 0, \\ \quad t(\sigma_1, s, \sigma_1', s') > 0 \wedge \Psi_2(s, \sigma_1, \sigma_2, s', \sigma_1', \sigma_2') \supset t(\sigma_2, s, \sigma_2', s') > 0, \\ \qquad \ldots \\ \qquad\quad t(\sigma_{n-1}, s, \sigma_{n-1}', s') > 0 \wedge \Psi_n(s, \sigma_{n-1}, \sigma_n, s', \sigma_{n-1}', \sigma_n) \supset t(\sigma_n, s, \sigma_n', s') > 0 \end{array}$$

such that every Ψ_i holds and $\sigma_n = \sigma$, $\sigma_n' = \sigma'$. We are now about to prove the second part of the proposition. The implication in the second part of the proposition is equivalent to the following two implications:

(1) $transPr(\sigma, s, \sigma', s') > 0 \wedge transPr(\sigma, s, \sigma'', s'') > 0 \wedge \sigma' \neq \sigma'' \supset$
$s' = do(tossHead, s) \wedge s'' = do(tossTail, s) \vee s' = do(tossTail, s) \wedge s'' = do(tossHead, s)$

(2) $transPr(\sigma, s, \sigma', s') > 0 \wedge transPr(\sigma, s, \sigma'', s'') > 0 \wedge s' \neq s'' \supset$
$s' = do(tossHead, s) \wedge s'' = do(tossTail, s) \vee s' = do(tossTail, s) \wedge s'' = do(tossHead, s)$.

We first show implication (1). As discussed above, $transPr(\sigma, s, \sigma', s') > 0$ implies that there is a sequence of implications involving σ_i, σ_i', Ψ_i, $i = 1, ..., n$ such that $\sigma = \sigma_n$ and $\sigma' = \sigma_n'$. Similarly, $transPr(\sigma, s, \sigma'', s'') > 0$ implies that there is another sequence of implications involving δ_i, δ_i', Ξ_i, $i = 1, ..., m$ such that $\sigma = \delta_m$ and $\sigma'' = \delta_m'$. Without loss of generality, let $m \geq n$. Then there is a base case which says $\Psi_1(\sigma_1, s, \sigma_1', s') \supset t(\sigma_1, s, \sigma_1', s') > 0$. Furthermore, by $\sigma' \neq \sigma''$ and (6.13), we get $\sigma_1 = \delta_{m-n+1}$ and $\sigma_1' \neq \delta_{m-n+1}'$ (per induction on n). As the different implications for sequences, conditionals, concurrent execution and procedures are mutually exclusive, Ψ_1 and Ξ_{m-n+1} must deal with the same (base) case, that is $n = m$ and the sequence involving δ_i, δ_i', Ξ_i starts with $\Xi_1(\sigma_1, s, \delta_1', s'') \supset t(\sigma_1, s, \delta_1', s'') > 0$.

Ψ_1 and Ξ_1 cannot deal with primitive actions or tests, because $\delta_1' \neq \sigma_1'$ holds by assumption, $\Psi_1(\sigma_1, s, \sigma_1', s')$ and $\Xi_1(\sigma_1, s, \delta_1', s'')$ must hold, and it is easy two verify that the conditions Ψ_i for primitive actions or tests satisfy the following:

$$\Psi_i(\sigma_1, s, \sigma_1', s') \wedge \Psi_i(\delta_1, s, \delta_1', s'') \wedge \sigma_1 = \delta_1 \supset \sigma_1' = \delta_1' = \mathsf{nil}.$$

Thus, the two implication involving Ψ_1 and Ξ_1 must be prob implications. As by assumption $\sigma_1' \neq \delta_1'$, the implication involving Ψ_1 must be an instance of the first implication for prob, and the implication involving Ξ_1 must be an instance of the second implication for prob, or vice versa. Thus, finally, we get $s' = do(tossHead, s) \wedge s'' = do(tossTail, s)$ or $s' = do(tossTail, s) \wedge s'' = do(tossHead, s)$, and hence the implication (1).

Next, we show implication (2). By a similar argument as above, there must be two sequences of implications involving σ_i, σ_i', Ψ_i respectively δ_i, δ_i', Ξ_i such that $\sigma = \delta_m = \sigma_n$. By (6.14), then, we get $\sigma_1 = \delta_1$, and two conditions $\Xi_1(\sigma_1, s, \sigma_1', s')$ and $\Psi_1(\sigma_1, s, \delta_1', s'')$ which must both deal with the same base case. It is easy two verify that the conditions Ψ_i for primitive actions or tests satisfy

$$\Psi_i(\sigma_1, s, \sigma_1', s') \wedge \Psi_i(\sigma_1, s, \delta_1', s'') \supset s' = s'' = s$$

and hence, by the assumption that $s' \neq s''$, Ψ_1 and Ξ_1 must belong to two prob implications. Thus, we get $s' = do(tossHead, s) \wedge s'' = do(tossTail, s)$ or $s' = do(tossTail, s) \wedge s'' = do(tossHead, s)$, and hence the thesis. □

The next proposition says that if $c_1, ..., c_n$ is a sequence of configurations such that there is a transition with positive weight from c_i to c_{i+1}, $i = 1, ..., n-1$, then for any $i, 1 \leq i \leq n$ the situation component of c_i is rooted in the situation component of c_1, and similarly the situation component of c_n is rooted in the situation component of c_i.

Proposition 14: *Let* $\mathrm{AX}_{\mathrm{transPr}}$ *be defined as above. Then for every model* M *of* $\mathrm{AX}_{\mathrm{transPr}}$*:*

- *If there exists a sequence* $\sigma_1, s_1, ..., \sigma_n, s_n$ *such that* $M \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ *for* $i = 1, ..., n-1$ *then* $M \models s_1 \sqsubseteq s_i \wedge s_i \sqsubseteq s_n$ *for* $i = 1, ..., n-1$*;*
- *In particular,* $M \models transPr^*(\sigma, s, \sigma', s') > 0 \supset s \sqsubseteq s'$*.*

Proof: Part (1) follows by induction of n, using Proposition 13 if $n > 1$. Part (2) follows from part(1) and Proposition 8 (page 100). □

The next proposition says that if two configurations c_n and c' can both be reached (by a sequence of transitions) from the same starting configuration c, and if the situation component of c_n can be obtained from the situation component of c' by a finite number of actions, then

the sequence of transitions from c to c' must be a prefix of the sequence of transitions from c to c_n.

Proposition 15: *Let* $\mathrm{AX}_{\mathrm{transPr}}$ *be defined as above. Then for every model* M *of* $\mathrm{AX}_{\mathrm{transPr}}$

- $M \models transPr^*(\sigma_1, s_1, \sigma_n, s_n) > 0 \wedge transPr^*(\sigma_1, s_1, \sigma', s') > 0 \wedge s' \sqsubset s_n$

implies that there exists a sequence $\sigma_1, s_1, ..., \sigma_n, s_n$ *such that*

- $M \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ *for* $i = 1, ...n-1$, *and*
- *there is a* $j, 1 \leq j < n$ *such that* $\sigma' = \sigma_j, s' = s_j$.

Proof: By contradiction. By Proposition 8 (page 100) there is a sequence $\sigma_1, s_1, ..., \sigma_n, s_n$ such that $M \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ for $i = 1, ...n-1$. So assuming that the implication does not hold means that there is no $j, 1 \leq j \leq n$ such that $\sigma' = \sigma_j, s' = s_j$.

Again by Proposition 8 we can conclude that there is a second sequence $\sigma'_1, s'_1, ..., \sigma'_m, s'_m$ from s_1, σ_1 to s', σ'. The sequence σ'_i, s'_i cannot be a superset of the sequence σ_i, s_i, because else we would get $s_n \sqsubseteq s'$ by Proposition 14; contradiction. Furthermore, the sequence σ'_i, s'_i must differ from the sequence σ_i, s_i, otherwise $\sigma' = \sigma_m, s' = s_m$. Let k be the smallest index such that $\sigma_k = \sigma'_k$ and $s_k = s'_k$ but $\sigma_{k+1} \neq \sigma'_{k+1}$ or $s_{k+1} \neq s'_{k+1}$. That is, $M \models transPr(\sigma_k, s_k, \sigma_{k+1}, s_{k+1}) > 0$ and $M \models transPr(\sigma_k, s_k, \sigma'_{k+1}, s'_{k+1}) > 0$. By Proposition 13 and the assumption $\neg(\sigma_{k+1} = \sigma'_{k+1} \wedge s_{k+1} = s'_{k+1})$, we get $s_{k+1} = do(tossHead, s_k) \wedge s'_{k+1} = do(tossTail, s_k)$ or vice versa. Thus, by Proposition 14 we get $do(tossHead, s_k) \sqsubseteq s'$ and $do(tossTail, s_k) \sqsubseteq s_n$, and hence the unique names axioms for situations imply $\neg(s' \sqsubseteq s_n)$. Contradiction. □

Finally, the next proposition shows that *doPr* is well-defined:

Proposition 16: *Let* $\mathrm{AX}_{\mathrm{transPr}}$ *be defined as above. Then:*

$$\mathrm{AX}_{\mathrm{transPr}} \models [transPr^*(\sigma, s, \delta, s') > 0 \wedge Final(\delta, s') \wedge transPr^*(\sigma, s, \delta', s') > 0 \wedge Final(\delta', s')] \supset \delta = \delta'.$$

Proof: By contradiction. Assume that there is an interpretation M of $\mathrm{AX}_{\mathrm{transPr}}$ and two configurations $\langle \delta, s' \rangle$ and $\langle \delta', s' \rangle$ such that $\delta \neq \delta'$ and $M \models transPr^*(\sigma, s, \delta, s') > 0$, $M \models transPr^*(\sigma, s, \delta', s') > 0$, $M \models Final(\delta, s')$ and $M \models Final(\delta', s')$.

By Proposition 14, we get $s \sqsubseteq s'$. Let us first consider the case where $s' = s$. By Proposition 8 we can conclude that there is a sequence of transitions seq from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$ and another sequence seq' from $\langle \sigma, s \rangle$ to $\langle \delta', s' \rangle$. Because $s' = s$, these transitions do not involve the execution of any new action, and in particular they do not involve any *tossHead* or *tossTail* action. By Proposition 13, then, we get that either seq is a subsequence of seq' or vice versa. Without loss of generality, we assume $\mathsf{seq} \subseteq \mathsf{seq}'$. Furthermore, $\mathsf{seq} \neq \mathsf{seq}'$ because of the assumption $\delta \neq \delta'$, hence there must be a transition with positive weight from $\langle \delta, s' \rangle$ to a successor configuration. At the same time, $\langle \delta, s' \rangle$ is assumed to be final. Contradiction with Proposition 12.

Let us now turn to the case $s \sqsubset s'$. Again, by Proposition 8 there is a sequence of transitions $\sigma_1, s_1, ..., \sigma_n, s_n$ from $\langle \sigma, s \rangle$ to $\langle \delta, s' \rangle$ and another sequence $\sigma'_1, s'_1, ..., \sigma'_m, s'_m$ from $\langle \sigma, s \rangle$ to $\langle \delta', s' \rangle$. The two sequences must differ because $\delta \neq \delta'$. Furthermore, by a similar

argument as above, none of the two sequences is a proper subsequence of the other. Thus the two sequences must differ at some point. Let k be the smallest index such that $\sigma_k = \sigma'_k$ and $s_k = s'_k$ but $\sigma_{k+1} \neq \sigma'_{k+1}$ or $s_{k+1} \neq s'_{k+1}$. That is, $M \models transPr(\sigma_k, s_k, \sigma_{k+1}, s_{k+1}) > 0$ and $M \models transPr(\sigma_k, s_k, \sigma'_{k+1}, s'_{k+1}) > 0$. By Proposition 13 and the assumption $\neg(\sigma_{k+1} = \sigma'_{k+1} \wedge s_{k+1} = s'_{k+1})$, we get $s_{k+1} = do(tossHead, s_k) \wedge s'_{k+1} = do(tossTail, s_k)$ or vice versa. Thus, by Proposition 14 we get $do(tossHead, s_k) \sqsubseteq s'$ and $do(tossTail, s_k) \sqsubseteq s'$. Contradiction with the unique names axioms for situations. □

6.2 A Control Architecture for Acting under Uncertainty

In order to reason about the possible effects of high-level controllers operating in uncertain domains, we have to adapt our model of the layered control architecture from Section 4.3. In particular, we have to account for the robot's uncertainty about the state of the world, and for the fact that low-level processes have uncertain outcomes. In this section, we a) show how to characterize the robot's beliefs about the state of the world in a probabilistic fashion, b) model noisy low-level processes by pGOLOG programs representing their different possible outcomes, and c) show how to deal with sensor processes like *inspect* which provide information about the state of the world, meaning that we integrate *sensing* into our architecture. The changes required to deal with uncertainty in our overall control architecture are illustrated in Figure 6.1.

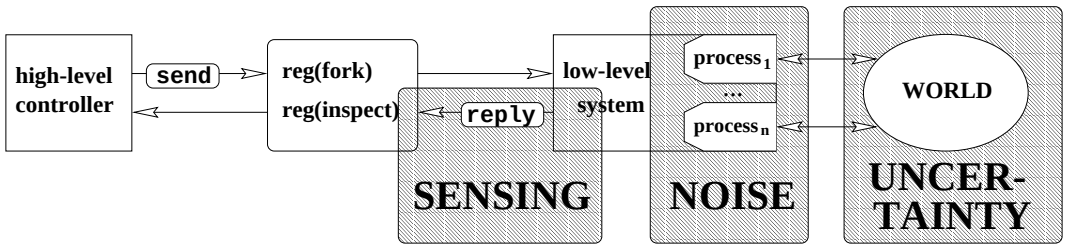


Figure 6.1: A Robot Control Architecture for Acting under Uncertainty

6.2.1 The Probabilistic Epistemic State

Let us begin with the representation of the robot's knowledge about the state of the world. While the plain situation calculus allows us to talk only about the actual world, in particular about the initial situation S_0, in scenarios like the ship/reject example, there is uncertainty about the initial situation. To deal with this uncertainty, we follow [BHL99] and characterize the probabilistic epistemic state of the robot by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities (cf. Section 3.3).

In our example, the robot initially considers two situations possible, s_1 and s_2, with degree of likelihood 0.3 and 0.7, respectively. As these two situations are the only situations considered possible, all other situations have likelihood 0. The following axiom makes this precise together with what holds and does not hold in each of the two situations:

$$\begin{aligned}
&\exists s_1, s_2 \forall s. s \neq s_1 \wedge s \neq s_2 \supset p(s, S_0) = 0 \wedge \\
&\quad p(s_1, S_0) = 0.3 \wedge p(s_2, S_0) = 0.7 \wedge \\
&\quad FL(s_1) \wedge BL(s_1) \wedge \neg PA(s_1) \wedge \neg PR(s_1) \wedge \neg ER(s_1) \wedge \\
&\quad \neg FL(s_2) \wedge \neg BL(s_2) \wedge \neg PA(s_2) \wedge \neg PR(s_2) \wedge \neg ER(s_2).
\end{aligned}$$

As described in Section 3.3.2, the term $Bel(\phi, s)$ is used to denote the agent's degree of belief that ϕ holds in situation s. (As usual, ϕ stands for a situation calculus formula where *now* is used to refer to the situation under consideration.) Formally, $Bel(\phi, s) = p$ is an abbreviation for the following term expressible in second-order logic:

$$\sum_{s':\phi[s']} p(s', s) / \sum_{s'} p(s', s) = p.$$

Intuitively, $Bel(\phi, s)$ is the normalized sum of the weights of all situations s' considered possible in s that fulfill ϕ; in our example, $Bel(FL, S_0)$ is 0.3.

6.2.2 The Communication between the High-Level Controller and the Low-Level Processes

Similar to Section 4.3.1, the communication between the high-level controller and the low-level processes is achieved through the fluent *reg* and the primitive actions *send* and *reply*. Arguably, there is no uncertainty about the value of the set of registers *reg*, i.e. the high-level controller has perfect information about them. Therefore, we require that *reg*'s initial value is reflected in all situations s' considered possible in S_0 (we will elaborate on the impact of this assumption in Section 6.2.4):

$$p(s', S_0) > 0 \supset reg(id, s') = reg(id, S_0).$$

For reasons of efficiency, we use a slightly different scheme than that employed in the previous chapter to represent the activation of the low-level processes. In Section 4.3.2, we made use of $reg(destRoom)$ and of an explicit test $(reg(destRoom) \neq currentRoom)?$ in the model of the navigation process to check whether the navigation process is activated. This activation scheme could easily be extended to a set of low-level processes by making use of a particular register r_p for each low-level process p. While this activation scheme is fine as long as the number of low-level processes involved is relatively small, it becomes expensive when the number of low-level processes becomes larger. In particular, during projection of a plan, in each simulated transition the projection mechanism has to evaluate each individual test.

To avoid this overhead, in this chapter we will proceed in a different manner. Instead of making use of a particular register r_p for each process p, we only use the special register *fork*.[4] The understanding is that whenever the high-level controller wants to activate a low-level process p, it assigns $reg(fork)$ the value p. For example, the high-level controller would execute $send(fork, paint)$ to tell the execution system to start the paint process. We remark that this activation scheme corresponds to an execution system where the different low-level processes are not continuously running, but instead are instantiated by the operating system when the high-level controller activates them.

Sensing and Sensor Processes While the high-level controller makes use of *send* actions to activate low-level processes, the low-level processes can execute *reply* actions to communicate with the high-level controller. In particular, the *reply* actions are used to provide information about the state of the world. For example, after its activation the *inspect* process will make use of physical sensors and appropriate pre-processing algorithms to estimate

[4] The term *fork* is an allusion to the procedure `fork` used in UNIX-like operating systems to create new concurrent processes.

whether or not the widget is blemished, whereupon it will execute a *reply*(*inspect*, *OK*) or a *reply*(*inspect*, $\overline{OK}$) action.[5] As a result, *reg*(*inspect*) is assigned the value *OK* respectively $\overline{OK}$. Thus, the *reply* actions are used to provide the high-level controller with an "*OK*" or "$\overline{OK}$" answer, i.e. with a (noisy) sensor reading.

We remark that the resulting view of sensing significantly differs from the well-known sensing actions of Scherl and Levesque [SL93, Lev96]. To us, sensing means: activate a special low-level process, which we sometimes call a *sensor process*. This "activation" has as an effect a *reply* action, a "sensor reading". We assume that every answer of a sensor process is provided by means of an exogenous *reply* action. Furthermore, we assume that the high-level controller is aware of all exogenous *reply* actions (as opposed to "actions" like *setPA* which are solely used to model the effects of the low-level processes). In short, sensing is thus realized by low-level processes which, after activation, cause an exogenous *reply* action to occur. In Chapter 7, we will show how *reply* actions are used to update the robot's belief state, that is, how they affect the set of situations considered possible.

6.2.3 Modeling the Low-Level Processes

While during real execution the actual sensor processes provide *reply* answers, for the task of projection we need a model specifying the possible ways the activation of a sensor process may result in a *reply* action. For example, we need a model of *inspect* specifying that if the widget is blemished then *inspect* will execute *reply*(*inspect*, $\overline{OK}$) with probability 90%. As mentioned in Section 6.1, we model all low-level processes by probabilistic pGOLOG programs. Thereby, however, we restrict the primitive actions involved in the pGOLOG model of a low-level process: *we do not allow the use of send actions*; see Appendix A.3.1 for the details. Intuitively, this restriction reflects the fact that only the high-level controller may execute *send* actions.

The following pGOLOG procedure simulates the effects of *inspect*, taking into account that *inspect* is only perfect if the widget is not blemished, and that otherwise there is a 10% probability to erroneously report *OK* although the widget is blemished. We assume that *inspect* first operates a camera for 7 seconds to get an image of the widget, and thereafter makes use of image processing routines which need another 3 seconds to provide an estimate. The program makes use of the procedure *waitTime* discussed below to represent the temporal extent of the low-level process:

$$
\begin{array}{l}
proc(inspectProc, [waitTime(7), \\
\qquad \mathsf{if}(PR, setER, \\
\qquad\quad \mathsf{if}(BL, [waitTime(3), \mathsf{prob}(0.9, reply(inspect, \overline{OK}), \\
\qquad\qquad\qquad\qquad\qquad\qquad reply(inspect, OK))], \\
\qquad\qquad\quad [waitTime(3), reply(inspect, OK)]))]).
\end{array}
$$

Intuitively $waitTime(n)$ waits for n seconds, that is, it remains blocked until *start* has increased by n. The following is a simple realization of $waitTime(n)$; at the end of this paragraph, we will see a more elegant definition of $waitTime(n)$. We assume a continuous fluent *clock*, which intuitively represents the actual time. Besides *clock*, we assume a functional fluent *timer*, which is set to the actual time by the primitive action *setTimer* and remains unmodified else. Formally:

[5] We remark that here *OK* and $\overline{OK}$ are ordinary terms, not logical formulas. We could as well use the terms 1 and 0.

$\mathsf{proc}(waitTime(n), [setTimer, waitFor(clock \geq timer + n)]);$

$$Poss(a, s) \supset timer(do(a, s)) = t \equiv a = setTimer \wedge start(s) = t \vee a \neq setTimer \wedge timer(s) = t;$$

$Poss(setTimer, s) \equiv \text{TRUE};$

$clock(S_0) = linear(0, 1, 0)$ and $Poss(a, s) \supset clock(do(a, s)) = clock(s)$.[6]

We remark that the conditional in *inspectProc* is evaluated 7 seconds after the activation of the process, reflecting the fact that the last 3 seconds are merely needed for the execution of image processing routines. Again, we stress that not all pGOLOG programs are meant to be actually executed. In fact, given the robot's uncertainty about the value of a fluent like *BL* it is unclear how the high-level controller should execute a pGOLOG program like *inspectProc* (similarly, the pGOLOG program *paintProc* cannot be executed by the high-level controller because it cannot directly execute actions like *setPA*). pGOLOG programs like the above are only used to reason about the effects of low-level processes; during real execution, the actual low-level processes are activated.

Next, let us model the other low-level processes. The following model of the *paint* process, which replaces the simpler one from Section 6.1, distinguishes two distinct steps: first, the widget is undercoated (UC), after which it is painted (PA). We assume that a possible blemish is removed immediately after activation of *paint*, and that after 10 seconds the widget is undercoated. After 30 seconds, the widget becomes painted with probability 95%. Finally, *paint* confirms completion by means of a $reply(painted, \top)$ action:[7]

$$\mathsf{proc}(paintProc, [clipBL, waitTime(10), \mathsf{if}(PR, setER, setUC), waitTime(20), \mathsf{if}(PR, setER, \mathsf{prob}(0.95, setPA)), reply(painted, \top)]).$$

Similarly, we model the remaining low-level processes *ship* and *reject*, accounting for their duration. The following two programs state that both *ship* and *reject* take 10 seconds to complete execution, whereupon they confirm completion by means of a $reply(processed, \top)$ action. Unlike *paint*, the low-level processes *ship* and *reject* have no uncertain outcomes, which means that we can specify their behavior without making use of prob instructions. However, *ship* and *reject* have conditional effects: if the widget is flawed, the execution of *ship* results in an error, and, vice versa, if it is not flawed, *reject* causes an error:

$\mathsf{proc}(shipProc, [waitTime(10), \mathsf{if}(PR \vee FL, setER), setPR, reply(processed, \top)]);$

$\mathsf{proc}(rejectProc, [waitTime(10), \mathsf{if}(PR \vee \neg FL, setER), setPR, reply(processed, \top)]).$

Aside - A More General Definition of *waitTime* As defined above, the procedure *waitTime* suffers from the drawback that it depends on the fact that no concurrent program affects the value of *timer*. This assumption is problematic for example if many programs modeling low-level processes are running concurrently. The following is a more general definition of *waitTime* which does not rely on the use of a fluent to memorize the time where it was activated. In the remainder of this thesis, whenever we write $waitTime(n)$ we refer to the following definition:

[6] Note that this ensures $val(clock(s), t) = t$.

[7] Strictly speaking, *paint* is thus also a sensor process: the $reply(painted, \top)$ action informs the high-level controller that the painting is completed.

$\mathsf{proc}(waitTime(n), waitUntil(start(now) + n));$

$\mathsf{proc}(waitUntil(t), [\text{TRUE}?, waitFor(clock \geq t)]).$

The new procedure $waitTime(n)$ is realized by a call to $waitUntil$ with $start(now) + n$ as argument. $waitUntil(t)$ simply waits until time t, using the continuous fluent *clock* introduced above. The reason why we need the dummy test TRUE? in *waitUntil* is quite subtle; the test is needed to ensure that the term $start(now)$ in the procedure call $waitUntil(start(now) + n)$ is evaluated as soon as possible *even if other programs are running concurrently.* That is, the dummy test ensures that $start(now)$ is evaluated as soon as the procedure $waitTime(n)$ is called. To see why the dummy test TRUE? is needed, let us consider the following program:

$$\{\mathsf{proc}(waitUntil_{naive}(t), waitFor(clock \geq t));\\ \mathsf{withPol}(waitFor(clock \geq 5), waitUntil_{naive}(start(now) + 10))\}.$$

An execution of the above program in situation S_0, assuming $start(S_0) = 0$, would result in $do([waitFor(clock \geq 5), waitFor(clock \geq 5 + 10)], S_0)$, i.e. in a situation with starting time 15 instead of 10! The reason is that the term $start(now)$ in the low-priority branch is actually re-evaluated after execution of the first *waitFor* action. The dummy test TRUE? in the body of *waitUntil* is used to avoid a similarly delayed evaluation of $start(now)$ if *waitTime* is called. This trick depends on the transition semantics of cc-Golog regarding procedure invocation (see Sections 4.2.1 and 3.2.3). The dummy test ensures that the execution of the procedure body of *waitUntil* starts as soon as possible, resulting in a transition to a new configuration which has the same situation component and a remaining program where $start(now)$ has already been evaluated *at transition time.*

The Overall Low-Level Execution System Given a characterization of every low-level process in terms of a pGOLOG program, we can characterize the behavior of the whole execution level by a single pGOLOG program. This program models the runtime-system of the robot, that is of the robot's operating system or "kernel process", which ensures that a *send* action results in the activation of the corresponding low-level process:

$$\begin{array}{l}\mathsf{proc}(kernelProc, [reg(fork) \neq nil?,\\ \quad \mathsf{if}(reg(fork) = inspect, [reply(fork, nil), \mathsf{withPol}(inspectProc, kernelProc)],\\ \quad\quad \mathsf{if}(reg(fork) = paint, [reply(fork, nil), \mathsf{withPol}(paintProc, kernelProc)],\\ \quad\quad\quad \mathsf{if}(reg(fork) = ship, [reply(fork, nil), \mathsf{withPol}(shipProc, kernelProc)],\\ \quad\quad\quad\quad \mathsf{if}(reg(fork) = reject, [reply(fork, nil), \mathsf{withPol}(rejectProc, kernelProc)],\\ \quad\quad\quad\quad\quad [reply(fork, nil), kernelProc]))))]).\end{array}$$

As long as $reg(fork)$ is nil, nothing happens. If $reg(fork)$ is assigned the name of a low-level process, then $reg(fork)$ is reset to nil, and the low-level process is run concurrently to the operating system's kernel process. In Section 7.2 (page 152), we will see that this activation scheme is also suited for the activation of low-level processes involving arguments (like the navigation process). We remark that similar to the procedure *navProc* used as a model of the low-level processes in the previous chapters (see page 72), *kernelProc* will run forever.

As an aside, we remark that we could as well stick to the activation scheme corresponding to the view where all processes are continuously active. In this case, we could use something like the following program as a model of the overall low-level execution system:

$$\begin{aligned}&\mathsf{conc}(\mathsf{forever}([\mathit{reg}(\mathit{paint}) = \mathsf{nil}?, \mathit{reply}(\mathit{paint}, \mathsf{nil}), \mathit{paintProc}]),\\ &\quad \mathsf{forever}([\mathit{reg}(\mathit{inspect}) = \mathsf{nil}?, \mathit{reply}(\mathit{inspect}, \mathsf{nil}), \mathit{inspectProc}]),\\ &\quad \mathsf{forever}([\mathit{reg}(\mathit{ship}) = \mathsf{nil}?, \mathit{reply}(\mathit{ship}, \mathsf{nil}), \mathit{shipProc}]),\\ &\quad \mathsf{forever}([\mathit{reg}(\mathit{reject}) = \mathsf{nil}?, \mathit{reply}(\mathit{reject}, \mathsf{nil}), \mathit{rejectProc}])).\end{aligned}$$

Note that the actual priority ordering of the different programs is arbitrary.

6.2.4 High-Level Plans and Directly Observable Fluents

While in ConGolog and cc-Golog plans the execution of actions is conditioned on the value of fluents which are either true or false, in a probabilistic setting the high-level controller has merely beliefs about the value of fluents. For example, in our example the robot initially believes that *FL* holds with probability 30%. As a result, in our probabilistic framework we no longer consider high-level plans whose tests appeal to the value of fluents, but instead consider *belief-based plans* which appeal to the robot's beliefs at execution time.[8]

In particular, we only consider programs written in a variant of pGOLOG which we call bGOLOG as legal high-level plans. bGOLOG programs are non-probabilistic and may only appeal to the robot's *beliefs*, that is to the term Bel which is a reified version of the fluent *Bel* (cf. Section 3.3.2) with situation argument suppressed. Similarly, a legal bGOLOG plan may not appeal to functional fluents in primitive actions or procedure calls (we will relax this assumption in Section 7.3.1). Furthermore, a bGOLOG plan may not include $\mathit{waitFor}(\tau)$ actions, because their execution appeals to the least time point of τ, and the robot is not guaranteed to know about it (we will elaborate on this assumption in Section 7.3.1). Finally, as we assume that the high-level controller cannot directly affect the state of the world (cf. Figure 6.1), we only consider bGOLOG programs as legal high-level plans if they do not execute any action which affects a fluent used to describe the state of the *world* (as opposed to a fluent like *reg* used to describe the state of the high-level controller). For simplicity, we restrict the primitive actions which may occur in a high-level bGOLOG plan to *send* actions. See Appendix A.3.2 for a thorough treatment of these issues, including the reification of Bel as first-order term. As the examples in the remainder of this chapter illustrate, these restrictions are not as severe as they may seem.

As a simple example, the following bGOLOG plan activates both *inspect* and *paint*, waits for their completion and finally processes the widget according to the result of *inspect*:

$$\begin{aligned}\Pi_{bb} \doteq [&\mathit{send}(\mathit{fork}, \mathit{inspect}), \mathsf{Bel}(\mathit{reg}(\mathit{inspect}) \neq \mathsf{nil}) = 1?,\\ &\mathit{send}(\mathit{fork}, \mathit{paint}), \mathsf{Bel}(\mathit{reg}(\mathit{painted}) \neq \mathsf{nil}) = 1?,\\ &\mathsf{if}(\mathsf{Bel}(\mathit{reg}(\mathit{inspect}) = \mathit{OK}) = 1, \mathit{send}(\mathit{fork}, \mathit{ship}), \mathit{send}(\mathit{fork}, \mathit{reject})),\\ &\mathsf{Bel}(\mathit{reg}(\mathit{processed}) \neq \mathsf{nil}) = 1?].\end{aligned}$$

The test $\mathsf{Bel}(\mathit{reg}(\mathit{inspect}) \neq \mathsf{nil}) = 1?$ can be used to verify that *inspect* has executed a *reply* action due to the fact that initially all registers have the value nil.[9] Thus, the value of $\mathit{reg}(\mathit{inspect})$ only changes from nil to a new value (i.e. *OK* or $\overline{\mathit{OK}}$) when *inspect* executes a $\mathit{reply}(\mathit{inspect}, \mathit{OK}/\overline{\mathit{OK}})$ action, and as the high-level controller is aware of all *reply* actions (cf.

[8] This is similar to Reiter's notion of knowledge-based programming [Rei00]. However, we remark that here we are dealing with *degrees of belief*.

[9] Note that actually $\mathsf{Bel}(\mathit{reg}(\mathit{inspect}) \neq \mathsf{nil}) = 1?$ is an abbreviation for $\mathsf{Bel}(\mathit{reg}(\mathit{inspect}, \mathit{now}) \neq \mathsf{nil}) = 1?$.

Section 6.2.2), its belief in $reg(inspect) \neq \mathsf{nil}$ rises to 1 right after the completion of *inspect*.[10] Similarly for the other epistemic tests.

Note that the above plan only appeals to the robot's beliefs concerning the value of the fluent *reg*. Furthermore, the robot's beliefs are only compared with the values 1 and 0, meaning that the above plan does not really appeal to probabilistic, real-valued beliefs. This is no coincidence. In fact, in this chapter we will only consider *pseudo-belief-based plans* whose tests and conditionals only appeal to the robot's beliefs concerning the set of registers *reg*. That is, while in general bGOLOG plans can also appeal to the robot's real-valued beliefs concerning other fluents, in this chapter we do not consider programs like the following:[11]

$$\begin{aligned}\Pi_{forbidden} \doteq [&send(fork, inspect), \mathsf{Bel}(FL) \neq 0.3?,\\ &send(fork, paint), \mathsf{Bel}(PA) > 0?,\\ &\mathsf{if}(\mathsf{Bel}(FL < 0.5, send(fork, ship), send(fork, reject)),\\ &\mathsf{Bel}(PR) = 1?].\end{aligned}$$

The reason why in this chapter we only consider epistemic tests appealing to the robot's beliefs concerning the set of registers *reg* is that in order to determine the belief in arbitrary sentences we need a successor state axiom for the fluent p, specifying how the robot's beliefs evolve over time. However, the specification of such a successor state axiom is quite complicated, and we will postpone it to the next chapter (we will consider general belief-based programs in Section 7.3.1). On the other hand, as we will see, it is straightforward to determine the robot's beliefs concerning the fluent *reg*.

Intuitively, the robot always has perfect information about the value of *reg*, because it is aware of all actions that affect its truth value, namely of all *send* and *reply* actions. As there is no uncertainty about the value of *reg*, we distinguish *reg* from other fluents and call it *directly observable*. Directly observable fluents are such that the agent always has perfect information about them - like the display of one's watch or a fuel gauge in the car. Formally, directly observables are defined as follows:

Definition 17 (Directly Observable Fluents) *Let Γ be a situation calculus theory including a characterization of an agent's epistemic state in terms of the fluent p (see Section 3.3). We call a relational fluent F* directly observable *with respect to Γ if and only if the following holds:*

$$\Gamma \models [S_0 \preceq s \wedge p(s', s) > 0] \supset F(\vec{x}, s') \equiv F(\vec{x}, s).$$

Similarly, we call a functional fluent f directly observable if and only if

$$\Gamma \models S_0 \preceq s \wedge p(s', s) > 0 \supset f(\vec{x}, s') = f(\vec{x}, s).$$

The following proposition, which follows directly from the definition of *Bel*, ensures that we do not have to worry about the fluent p to determine the robot's beliefs concerning the value of directly observable fluents.

Proposition 18:

Let Γ be a situation calculus theory including a characterization of an agent's epistemic state in terms of the fluent p, and F a directly observable fluent with respect to Γ. Then:

[10] We remark that so far, this claim is only supported by our intuition, as we have not yet presented a successor state axiom for p, specifying how the robot's beliefs evolve over time.

[11] We remark that in the ship/reject domain $\Pi_{forbidden}$ specifies the same behavior as Π_{bb}.

$$\Gamma \models S_0 \preceq s \supset [Bel(F(\vec{x}), s) = 1 \wedge F(\vec{x}, s) \vee Bel(F(\vec{x}), s) = 0 \wedge \neg F(\vec{x}, s)]$$

Proof: (Outline) In calculating $Bel(F(\vec{x}), s)$, it is sufficent to consider the situations s' with positive weight, that is $Bel(F(\vec{x}), s) = \sum_{\{s':F(\vec{x},s')\wedge p(s',s)>0\}} p(s', s) / \sum_{\{s':p(s',s)>0\}} p(s', s)$. Furthermore, F is directly observable, hence $S_0 \preceq s \supset [p(s', s) > 0 \supset F(\vec{x}, s') \equiv F(\vec{x}, s)]$. Thus,

$$S_0 \preceq s \supset Bel(F(\vec{x}, s) = p \equiv \frac{\sum_{\{s':F(\vec{x},s')\wedge p(s',s)>0\}} p(s', s)}{\sum_{\{s':p(s',s)>0\}} p(s', s)} = p$$

is equivalent to the following expression, where s' is replaced by s in the condition $F(\vec{x}, s)$:

$$S_0 \preceq s \supset Bel(F(\vec{x}, s) = p \equiv \frac{\sum_{\{s':F(\vec{x},s)\wedge p(s',s)>0\}} p(s', s)}{\sum_{\{s':p(s',s)>0\}} p(s', s)} = p$$

and thus to

$$S_0 \preceq s \supset Bel(F(\vec{x}, s) = p \equiv [F(\vec{x}, s) \wedge \frac{\sum_{\{s':p(s',s)>0\}} p(s', s)}{\sum_{\{s':p(s',s)>0\}} p(s', s)} = p \vee \neg F(\vec{x}, s) \wedge p = 0],$$

which is equivalent to $S_0 \preceq s \supset Bel(F(\vec{x}, s) = p \equiv [F(\vec{x}, s) \wedge p = 1 \vee \neg F(\vec{x}, s) \wedge p = 0]$. Hence, the proposition holds. □

As we have already discussed in Section 6.2.2, in our model of the overall control architecture there is no uncertainty about the initial value of *reg*. Furthermore, we have required that its initial value is reflected in S_0 (cf. Section 6.2.2). This, together with the facts that *reg* changes its value only as a result of the primitive actions *send* and *reply* and that we assume that the high-level controller is aware of all *send* and *reply* actions, implies that in our overall control architecture the fluent *reg* is directly observable. (As we do not provide a successor state axiom for the fluent p in this chapter, we have to postpone a formal proof of this claim to Section 7.1.5, Proposition 25.)

Thus, for the execution of pseudo-belief-based plans that only appeal to the robot's beliefs concerning the set of registers *reg*, we do not need to consider the fluent p and the set of situations considered possible. The whole reasoning can be based solely on the actual state of the world, as we do not have to distinguish between the value of the fluent *reg* and robot's beliefs about *reg*. In particular, this means that we do not have to worry about the robot's epistemic state during the actual execution of a bGOLOG plan appealing only to the robot's beliefs concerning *reg*.[12] We remark that as a result, in our framework directly observable fluents play a similar role as the *program variables* used in robot programming languages like RPL.

6.3 Probabilistic Projection in pGOLOG

Now that we have introduced pGOLOG and have presented our extended control architecture for acting and reasoning under uncertainty, we can turn to the probabilistic projection of a bGOLOG plan: How probable, from the point of view of a robot with a probabilistic belief state, is it that a sentence ϕ will hold *after* the execution of the bGOLOG plan σ in situation s, given a faithful characterization of the low-level execution system in terms of a pGOLOG program?

[12] This is unlike [Lak99], which makes use of the special term `Kwhether` to ensure that the truth value of fluents tested within a plan are *known* at execution time.

6.3.1 Projected Belief

To determine the projected belief that a sentence ϕ will hold after execution of a bGOLOG plan σ in situation s, we have to simulate the concurrent execution of the plan σ and the pGOLOG model of the low-level processes, similar to Section 4.3.3. We have already seen in Section 6.1.2 that the execution of a probabilistic pGOLOG program may result in different execution traces, which all have to be considered to determine the probability that a sentence holds afterwards. However, in Section 6.1.2 we only considered the execution of a plan *in one specific situation*. Here, we are dealing with a probabilistic epistemic state represented by a *set* of possible situations. For example, in the ship/reject domain our robot initially considers two situations s_1 and s_2 possible (cf. Section 6.2.1). In order to correctly determine the projected belief in ϕ, we thus have to *simulate the execution of* σ (which may result in different execution traces) *in every situation* s' *considered possible in* s, weighted by the likelihood of s' in s as specified by $p(s', s)$.

We are now ready for the formal definition of projected belief. Let ϕ be a formula whose only term of sort situation is the special situation term *now*, s a situation, ll_{model} a model of the low-level processes (for example *kernelProc*), and σ a bGOLOG program. We write $PBel(\phi, s, \sigma, ll_{model})$ to denote the agent's belief that ϕ holds after the execution of σ in situation s. $PBel(\phi, s, \sigma, ll_{model}) = p$ is an abbreviation for the following term expressible in second-order logic:

$$\frac{\sum_{s', s^*:\phi[s^*]} p(s', s) * doPr(\mathsf{withPol}(ll_{model}, \sigma), s', s^*)}{\sum_{s'} p(s', s)} = p.$$

As usual, $\phi[s^*]$ denotes the formula obtained by substituting the situation variable s^* for all occurrences of *now* in fluents appearing in ϕ. $PBel(\phi, s, \sigma, ll_{model})$ is thus defined to be the weight of all paths that reach a final configuration $\langle \delta^*, s^* \rangle$ where $\phi[s^*]$ holds, starting from a configuration $\langle \mathsf{withPol}(ll_{model}, \sigma), s' \rangle$, weighted by the robot's belief in s'. Note that the low-level processes are taken into account by concurrently executing ll_{model}. As usual, we will leave out the *now* argument when no confusion arises.

In order to correctly project bGOLOG plans appealing to the robot's beliefs, we assume that the accessibility relation defined by the fluent p is *Euclidean*, meaning that for any legal situation s and any situation s' considered possible in s, s and s' agree on the set of situation considered possible. Formally, we require the following axiom:[13]

$$S_0 \preceq s \wedge p(s', s) > 0 \wedge p(s'', s) = p \supset p(s'', s') = p.$$

We remark that there are configurations $\langle \sigma, s \rangle$ such that $PBel(\text{TRUE}, s, \sigma, ll_{model}) < 1$. This is due to the fact that (probabilistic branches of) the program $\mathsf{withPol}(ll_{model}, \sigma)$ may never become final. For example, the agent's projected belief that TRUE? holds after the execution of a program which results in the simulation of a program like $\mathsf{prob}(0.5, \text{FALSE?}, \text{TRUE?})$ is merely 50%. Possibly *PBel* may be better read as "the probability that σ ends and that thereafter ϕ holds".

6.3.2 Examples

To illustrate our definition of *PBel*, we will now consider probabilistic projections of some simple plans in the ship/reject domain.

[13] We remark that so far we have not defined a successor state axiom for p. We will reconsider the assumption that p is Euclidean in Section 7.1.5, where we specify how p evolves from one situation to another.

A simple plan We begin with the simple plan Π_{robby1}, which activates the *paint* process, and ships the widget after *paint* has finished execution. For readability, we use *processed* as a shorthand for $\mathsf{Bel}(reg(processed) = \top) = 1$, *painted* as a shorthand for $\mathsf{Bel}(reg(painted) = \top) = 1$, *forkInspect* as a shorthand for $send(fork, inspect)$, and similarly for *forkPaint*, *forkShip* and *forkInspect*:

$$\Pi_{robby1} \doteq [forkPaint, painted?, forkShip, processed?].$$

We will now calculate the robot's projected belief that the widget is painted, processed and that no execution error has occurred after the execution of Π_{robby1} in S_0, using *kernelProc* as the model of the low-level processes. Let $\mathrm{AX}_{\mathrm{pGolog}}$ be the set of axioms $\mathrm{AX}_{\mathrm{transPr}}$ (cf. Proposition 11) together with the set of axioms $\mathrm{AX}_{\mathrm{arch}}$ and the definition of *doPr*. Furthermore, let Γ be the set of axioms $\mathrm{AX}_{\mathrm{pGolog}}$ together with successor state axioms for the fluents *PA*, *PR*, *ER*, *FL* and *BL*, *val* axioms for the *t-function*s (in our example scenario, we only need the *t-function linear*), axioms for the continuous fluent *clock*, precondition axioms stating that all *set* and *clip* actions are always possible, the axiom from Section 6.2.1 specifying the initial belief state, the above axiom stating that p is Euclidean, and the facts $S_0 \preceq s \supset [Bel(reg(id) = val), s) = 1 \equiv reg(id, s) = val]$, stating that *reg* is directly observable, and $reg(id, S_0) = \mathsf{nil}$, our usual convention that initially all registers have value nil. Then, it is possible to show:

$$\Gamma \models PBel(PA \wedge PR \wedge \neg ER, S_0, \Pi_{robby1}, kernelProc) = 0.665.$$

Note that although Γ does not include a successor state axiom for p, we can simulate the execution of bGOLOG plans like Π_{robby1}, appealing only to the robot's beliefs with respect to *reg*, because *reg* is directly observable. The projected belief is determined as follows: if the world is as described by s_1 (cf. Section 6.2.1), that is if the widget is flawed, then the execution of *ship* will result in an error, and the overall plan will fail regardless of the outcome of *paint*. On the other hand, if the world is as described by s_2, the widget will be processed correctly. However, there is a 5% probability that the *paint* process will not make *PA* true. Given that the initial probability of s_2 is 0.7, this results in a total success probability of $0.7 * 0.95 = 0.665$.

In fact, we can deduce that the concurrent execution of *kernelProc* and Π_{robby1}, starting in a situation s' considered possible in S_0, can result in one of four possible execution traces s^*. In the following formula, we write $waitFor(t)$ as an abbreviation for $waitFor(clock \geq t)$. To facilitate the readability, we have underlined actions executed by the high-level controller:

$$\begin{aligned}
&\Gamma \models p(s', S_0) > 0 \wedge doPr(\mathsf{withPol}(kernelProc, \Pi_{robby1}), s', s^*) > 0 \equiv \\
&\quad \exists s_0'.[s^* = do([\underline{send(fork, paint)}, reply(fork, \mathsf{nil}), clipBL, waitFor(10), setUC, \\
&\qquad\qquad \underline{waitFor(30)}, tossHead, setPA, reply(painted, \top), \underline{send(fork, ship)}, \\
&\qquad\qquad reply(fork, \mathsf{nil}), waitFor(40), setER, setPR, reply(\underline{processed}, \top)], s_0') \vee \\
&\qquad s^* = do([\underline{send(fork, paint)}, reply(fork, \mathsf{nil}), clipBL, waitFor(10), setUC, \\
&\qquad\qquad \underline{waitFor(30)}, tossTail, reply(painted, \top), \underline{send(fork, ship)}, \\
&\qquad\qquad reply(fork, \mathsf{nil}), waitFor(40), setER, setPR, reply(\underline{processed}, \top)], s_0') \vee \\
&\qquad s^* = do([\underline{send(fork, paint)}, reply(fork, \mathsf{nil}), clipBL, waitFor(10), setUC, \\
&\qquad\qquad \underline{waitFor(30)}, tossHead, setPA, reply(painted, \top), \underline{send(fork, ship)}, \\
&\qquad\qquad reply(fork, \mathsf{nil}), waitFor(40), setPR, reply(\underline{processed}, \top)], s_0') \vee \\
&\qquad s^* = do([\underline{send(fork, paint)}, reply(fork, \mathsf{nil}), clipBL, waitFor(10), setUC,
\end{aligned}$$

$$waitFor(30), tossTail, reply(painted, \top), \underline{send(fork, ship)},$$
$$reply(fork, \mathsf{nil}), waitFor(40), setPR, reply(processed, \top)], s'_0)].$$

Proof: Similar to the proof in Section 6.1.2, this is straightforward but laborious. We only give an intuitive sketch. Besides, we remark that the above execution traces were also obtained by running the pGOLOG interpreter from Section 8.3 on the example (as well as on the other examples in this chapter).

The first two disjuncts represent the possible execution traces if the world is as described by s_1; the last two represent the possible execution traces if the world is as described by s_2. We will only consider the first two execution traces. They both start with a $send(fork, paint)$ action executed by Π_{robby1}. This action has the effect of unblocking *kernelProc*, which first executes $reply(fork, \mathsf{nil})$ and then invokes the procedure *paintProc*. *paintProc*'s first action is *clipBL*. Thereafter, it waits for 10 seconds. During this time, the high-level plan is blocked, because it is waiting for $Bel(reg(painted) = \top) = 1$ to become true. After waiting 10 seconds, *paintProc* (or rather what remains of it) resumes execution. *PR* does not hold, so it executes *setUC*, and then waits for another 20 seconds. Thereafter, it executes the prob instruction. Recall that the two probabilistic branches represent, respectively, the case where the *paint* process successfully causes the widget to be painted, or fails. If the first branch if taken, this results in the execution of *setPA*. Thereafter, the *paint* process executes $reply(painted, \top)$ and finishes execution. This case corresponds to the first disjunct in the above statement. If the other branch is taken, the paint process immediately executes $reply(painted, \top)$. This case corresponds to the second disjunct in the above axiom.

From here, the two execution traces (the first and second disjunct) evolve alike. The $reply(painted, \top)$ action unblocks the high-level plan, which then executes $send(fork, ship)$. As before, this action unblocks *kernelProc*, which first executes $reply(fork, \mathsf{nil})$ and then invokes the procedure *shipProc*. The high-level plan is blocked, waiting for $Bel(reg(processed) = \top)$ to become 1, and thus *shipProc* first executes $waitTime(10)$. Thereafter, it executes the conditional $\mathsf{if}(PR \vee FL, setER)$. *FL* holds (recall that we consider the execution traces that result if the world is as described by s_1), and thus the pGOLOG model of *ship* executes *setER*. Finally, it executes *setPR* and $reply(processed, \top)$, which unblocks the high-level plan and completes the execution.

The last two disjunct can be obtained analogously. Note that they do not include *setER* actions because they represent the possible execution traces if the world is as described by s_2, meaning that *FL* does not hold. □

We remark that the test *processed*? at the end of Π_{robby1} is necessary to obtain correct results. In fact, the projection of $[forkPaint, painted?, forkShip]$ results in the widget being processed with probability 0, because the condition *PR* is tested right after the activation of *ship*, without waiting for its completion.

A Sensing Plan Next, we will project the very simple sensing plan Π_{robby2} which activates the *inspect* process and waits until *inspect* provides an answer. We write *inspected* as a shorthand for $\mathsf{Bel}(reg(inspect) \neq \mathsf{nil}) = 1$:

$$\Pi_{robby2} \doteq [forkInspect, inspected?].$$

Let us consider the projected belief in *FL* after the execution of this plan. Let Γ be defined as above. Then one can deduce:

$$\Gamma \models PBel(FL, S_0, \Pi_{robby2}, kernelProc) = 0.3.$$

The projected uncertainty in the widget being flawed does not differ from the uncertainty in S_0! Although this might at first come as a surprise, it is indeed the only possible result, because the *projection* of a sensing action can by no means sharpen the robot's belief state (the actual execution, of course, can do so because it provides an actual observation). Actually, the effect of *inspect* is to modify the value of *reg*(*inspect*), making it an estimate for the value of *BL*. This effect is reflected in the projected belief state:

$$\begin{aligned}\Gamma \models\ & p(s', S_0) > 0 \wedge doPr(\mathsf{withPol}(kernelProc, \Pi_{robby2}), s', s^*) = p \wedge p > 0 \equiv \\ & BL(s^*) \wedge reg(inspect, s^*) = \overline{OK} \wedge p = 0.3 * 0.9 \vee \\ & BL(s^*) \wedge reg(inspect, s^*) = OK \wedge p = 0.3 * 0.1 \vee \\ & \neg BL(s^*) \wedge reg(inspect, s^*) = OK \wedge p = 0.7.\end{aligned}$$

Proof: (Outline) In all execution traces, Π_{robby2} executes *send*(*fork*, *inspect*) which activates *inspectProc*, the model of the inspect process. The first two execution traces s^* characterized above correspond to the case where the world is as described by s_1, meaning that both *FL* and *BL* holds. In this case, *inspectProc* can either execute a *reply*(*inspect*,$\overline{OK}$) or a *reply*(*inspect*,*OK*), with probability 90% respectively 10%. As the probability that the world is as in s_1 is 30%, the respective weight of the two execution traces if 0.3*0.9 respectively 0.3*0.1. Finally, the last execution trace corresponds to the case where the world is as described by s_2. The probability that the world is as in s_1 is 70%, and there is only one possible execution of *inspectProc*, which involves a *reply*(*inspect*,*OK*) action. This execution trace has probability 0.7*1.0. □

As we can see, there is a correlation between the value of *reg*(*inspect*) and the truth value of *BL*. In particular, we can deduce that the directly observable fluent *reg*(*inspect*) is a correct estimate of the value of *BL* with probability 97%:

$$\Gamma \models PBel([BL \equiv (reg(inspect) = \overline{OK})], S_0, \Pi_{robby2}, kernelProc) = 0.97.$$

Roughly, this is because there is only one possible execution trace s^* where $[BL(s^*) \equiv (reg(inspect, s^*) = \overline{OK})]$ does not hold, and this situation is quite unlikely: it has only probability 0.3*0.1.

A Conditional Plan The above result suggests that a conditional plan branching on the value of *reg*(*inspect*), or, more exactly, on the robot's beliefs regarding *reg*(*inspect*), can achieve a higher probability to correctly process the widget than the unconditional plan Π_{robby1}. As an example, the following example plan Π_{robby3} combines sensing with the conditional activation of *ship* respectively *reject* in order to achieve $PR \wedge \neg ER$. We write *OK* as a a shorthand for $\mathsf{Bel}(reg(inspect) = OK)) = 1$:

$$\Pi_{robby3} \doteq [forkInspect, inspected?, \mathsf{if}(OK, forkShip, forkReject), processed?].$$

From Γ, we can deduce that this conditional plan has a higher probability to achieve $PR \wedge \neg ER$ than the non-conditional plan Π_{robby1}:

$$\begin{aligned}\Gamma \models & PBel(PR \wedge \neg ER, S_0, \Pi_{robby3}, kernelProc) = 0.97 \wedge \\ & PBel(PR \wedge \neg ER, S_0, \Pi_{robby1}, kernelProc) = 0.7.\end{aligned}$$

The reason for the superiority of the conditional plan is the correlation of $(reg(inspect) = \overline{OK})$ and FL after execution of *inspect*.

A solution to the example problem We end this section with the projection of a conditional plan that represents a reasonable solution to the ship/reject example. This plan, Π_{robby4}, is similar to the above plan Π_{robby3} but additionally activates the *paint* process before branching on $reg(inspect)$. We remark that the following plan Π_{robby4} coincides with the example plan Π_{bb} from Section 6.2.4:

$$\begin{aligned}\Pi_{robby4} \doteq [&forkInspect, inspected?, forkPaint, painted?\\ &\mathsf{if}(OK, forkShip, forkReject), processed?].\end{aligned}$$

This plan has a probability of 92.15 % to achieve the goal to paint and correctly process the widget:

$$\Gamma \models PBel(PA \wedge PR \ \wedge \neg ER, S_0, \Pi_{robby4}) = 0.9215.$$

6.3.3 Probabilistic Projection and Expected Utility

So far, we have only considered the question "how probable is it that a sentence ϕ will hold after the execution of the plan σ". In examples like the ship/reject domain where a robot is to fulfill a given goal with a certain probability, this is all there is to answer. In other settings, however, a robot is often concerned with the goal to maximize the *expected utility* of its behavior, a common notion in utility theory and decision theory (cf. [RN95]). Here the idea is that the agent is given a real valued, bounded *reward function* which assigns a specific reward to every state or situation, and has the goal to maximizes the expected total accumulated reward over some horizon of interest. The key difference compared to the issues we considered so far is that the quality of a possible plan is determined by its expected reward, rather than by its probability to achieve a sentence ϕ.

It turns out that the concepts of utility theory can easily be integrated into our framework. To illustrate how expected utility can be dealt with in our framework, we follow [BRST00] and make use of the functional fluent $reward(do(a, s))$ of sort *Real* that asserts costs and rewards to each situation as a function of the action taken, properties of the current situation, or both. For example, we might assert that each activation of *paint* and *inspect* causes costs of 1 unit, shipping a widget which is painted and not flawed provides a reward of 10 units, but erroneously shipping a widget which is flawed or not painted causes a cost of 20 units:

- $reward(do(send(fork, inspect), s)) = -1$
- $reward(do(send(fork, paint), s)) = -1$
- $reward(do(send(fork, ship), s)) =$ **if** $\neg FL(s) \wedge PA(s)$ **then** 10 **else** -20.

We assume that all other actions result in reward 0. In order to calculate the total utility of a possible execution trace, we then define the functional fluent U of sort *Real* which has value 0 in all initial situations, and which changes its value as a result of the execution of primitive actions:

$$\begin{aligned}&\forall s'.p(s', S_0) > 0 \supset U(s') = 0;\\ &Poss(a) \supset [U(do(a, s)) = u \equiv U(s) = u' \wedge reward(do(a, s)) = r \wedge u = u' + r].\end{aligned}$$

$EU(\sigma, s, ll_{model})$, the expected utility of the plan σ in situation s given ll_{model} as model of the low-level processes, is then defined as the weighted average of the value of U in all possible execution traces resulting from the execution of σ in s. Formally, the expression $u = EU(\sigma, s, ll_{model})$ is an abbreviation for the following formula expressible in second-order logic:[14]

$$u = \sum_x x * PBel(U(now) = x, s, \sigma, ll_{model}).$$

Note the use of *PBel* in the definition *EU*. As an example, let us determine the expected utility of the plan Π_{robby4} from Section 6.3.2. Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{pGolog}}$ from Section 6.1.2 together with successor state axioms for the fluents *PA*, *PR*, *ER*, *FL* and *BL*, *val* axioms for the *t-function*s, axioms for the continuous fluent *clock*, precondition axioms stating that all set and clip actions are always possible, the axiom from Section 6.2.1 specifying the initial belief state and finally appropriate axioms for the functions *reward* and *U* sketched above. Then, it is possible to show:

$$\Gamma \models EU(\Pi_{robby4}, S_0, kernelProc) = 3.35.$$

The expected utility is determined as follows: in any case, *paint* and *inspect* are activated once, causing a total cost of 2 units. The probability to ship a widget which is not flawed and painted is $70\% * 95\%$. The probability to ship a widget which is not flawed but neither painted is $70\% * 5\%$. Finally, the probability to erroneously ship a flawed widget is $30\% * 10\%$. The total expected utility is thus $-1 - 1 + 10 * 0.7 * 0.95 - 20 * 0.7 * 0.05 - 20 * 0.3 * 0.1 = 3.35$.

6.4 Discussion

Summarizing, we have proposed pGOLOG, a probabilistic extension of GOLOG, as a modeling language for noisy low-level processes. Then, we have shown how to extend the control architecture from Section 4.3 to account for the robot's uncertainty about the state of the world. Next, we have introduced sensor processes as a means to handle sensing within this architecture, and have shown how pGOLOG can be used to model all low-level processes involved. Thereafter, we have defined bGOLOG plans, and have introduced the notion of directly observable fluents. In bGOLOG plans, directly observable fluents play a role similar to program variable in ordinary programming languages. Next, we have provided a projection mechanism that allows us to assess the probability that a sentence will hold after the execution of a bGOLOG plan. The integration of uncertainty and sensing in our framework allows us to formally verify the superiority of conditional, sensing plans over simple sequential plans. Finally, we have sketched the relation between projection and expected utility.

Before we compare pGOLOG with some related approaches, we make a note on pGOLOG's relation to nondeterminism. Unlike in ConGolog, dealing with nondeterministic instructions like $(\sigma_1|\sigma_2)$ (nondeterministic choice) or $\pi x.\sigma$ (nondeterministic choice of argument) is problematic in pGOLOG. This is due to the fact that both nondeterministic instructions and probabilistic prob instructions can result in multiple possible successor configurations. In particular, the semantics of ConGolog's nondeterministic instructions is defined by means of *Trans* axioms which specify more than one successor configuration. Similarly, pGOLOG's *transPr* definition

[14] In utility theory, it is quite common to accumulate the rewards only up to a finite time horizon. We remark that within our framework this could be realized similarly to the definition of the limited lookahead construct of Section 5.2.2.

concerning prob also specifies two possible successors, but *additionally assigns a weight to the different possible successors*. The problem is that nondeterministic ConGolog and pGOLOG use the feature of multiple possible outcomes for different purposes. In the probabilistic case, the different outcomes of a prob instruction have specific probabilities, and these probabilities are used during probabilistic projection where the weights of different branches are summed up. On the other hand, in the nondeterministic case the different outcomes are not associated a weight, and the interpreter is not counting different outcomes but is just looking for a successful execution trace. It is not clear how to handle both kind of nondeterministic outcomes directly within the definition of *Trans* respectively of *transPr*.[15]

The work of Bacchus, Halpern and Levesque [BHL95, BHL99] on noisy sensors and effectors may seem like an alternative to our treatment of probabilistic outcomes. However, although technically the work presented in this chapter is based on concepts like the p fluent introduced by BHL, the topic of their approach and of the approach presented in this chapter are different. They are concerned with how the epistemic state of an agent changes as a result of the execution of noisy actions and the perception of noisy sensor readings, an aspect we completely ignore in this chapter (we will investigate this issue in Chapter 7). In turn, they do not consider the probabilistic projection of a plan, and it is not clear how their framework could be extended to allow for projection. Instead, we model noisy low-level processes as probabilistic procedures, and make use of these procedures to *project* the (prior) probability that a sentence will hold after the execution of a bGOLOG plan. Note that the tasks of probabilistic projection and belief update are different in nature because the former does not involve any actual execution or sensing information, while the latter deals with actual execution and sensing information.

Maybe the closest work to that reported in this chapter is Poole's [Poo96, Poo98] integration of independent choice logic [Poo97], the situation calculus and conditional plans. Similar to us, Poole aims at representing and reasoning about the different outcomes of conditional plans executed in stochastic domains. In particular, Poole is concerned about the expected utility of a plan. Although we focus on probabilistic projection and only briefly consider expected utility, Poole's and our framework are both concerned with predicting the probabilistic effects of a plan, as opposed to BHL which consider the way the belief of an agent changes during execution. Poole's view of sensing is also quite similar to ours: he assumes passive sensors (i.e. no sensing actions as in Scherl and Levesque [SL93, Lev96]), which are represented by means of a set of terms called *observables*. In particular, in any situation the robot has perfect knowledge of the value of the observables. This is quite similar to our notion of directly observable fluents. However, the technical design decisions are different in Poole's work and in ours. In particular, we differ with respect to whether a situation should determine what facts hold at a point, that is in what state the world is in.[16] In our framework, we keep the conceptualization that a situation corresponds to a state (however, the agent doesn't know what situation it is in). In Poole's framework, an agent knows what situation it is in, but the situation doesn't fully specify what is true.[17] Besides, in Poole's framework actions are equated with motor control commands that are sent by the agent's controller, while in

[15] A possible approach to handling nondeterminism in pGOLOG is proposed in [GL00b], where a nondeterministic plan is mapped to deterministic variants of the plan, which are amenable to probabilistic projection.

[16] The distinction of state and situation made here is somewhat similar to that underlying the fluent calculus [Thi99c].

[17] In particular, there is a probability distribution over which facts hold in the different situations. To determine the truth value of a fluent in a situation, one has to make use of a *selector function* [Poo97].

our approach actions are used to represent arbitrary change. Finally, while Poole's approach is based on *stable model* semantics [GL88], our framework is based on the standard predicate calculus semantics.

Acting under uncertainty lies at the heart of POMDPs [KLC98] and they deal with these aspects in a more exhaustive way, but the computational cost is prohibitive already in relatively small domains (e.g. [GB98]). Note that unlike POMDPs, probabilistic planners [KHW95, ML98], and probabilistic planners that accounts for sensing [GA99, BL99], our framework is fully logic based and much more expressive since we are not restricted to propositional representations. On the other hand, the recently proposed DTGolog [BRST00, Sou01] assumes full observability of the domain. As for high-level programming approaches like [Lev96, Lak99, Rei00] which are based on Scherl and Levesque's representation of (incomplete) knowledge and sensing in the situation calculus [SL93], they do not account for probabilistic uncertainty. Finally, we remark that all the related approaches mentioned in this section do not account for the temporal extent of low-level processes.

Chapter 7

Belief Update in pGOLOG

In the previous chapter, we have presented an extension of the GOLOG framework which allows the *probabilistic projection* of plans, based on a probabilistic characterization of the robot's belief state and of the noisy low-level processes. So far, however, we have only considered the projection of a plan based on an explicit specification of the actual epistemic state. In particular, we have only considered projection in the initial situation, based on a specification of the initial epistemic state of the robot. However, similar to cc-Golog where we allowed on-the-fly projection during on-line execution, we would like to interleave on-line execution and probabilistic projection in pGOLOG. In order to allow probabilistic projection during on-line execution, a robot controller must adapt its beliefs during the execution of a plan involving the activation of noisy low-level (sensor) processes. We will refer to this task as *belief update*, following Bacchus, Halpern and Levesque [BHL99]. Belief update is not only a prerequisite for the definition of local lookahead constructors which allows probabilistic projection under user control, but also allows the execution of *belief-based programs*, high-level programs that appeal to the agent's real-valued beliefs *at execution time*.

To get a better intuition for the difference between belief update and probabilistic projection, let us consider some examples in the ship/reject domain introduced in the previous chapter. In this domain a typical projection task is the following: "how probable is it that the plan $[inspect, \mathsf{if}(OK, ship, reject)]$ will falsely ship a flawed widget?" On the other hand, belief update is concerned with questions like: "what is the probability that the widget is flawed if during on-line-execution the robot actually perceived OK?[1]" The difference between the two tasks is that in the former case, the agent reasons about how the world might evolve, while in the latter case its beliefs change as a result of actual actions. Finally, a belief-based plan is a specification appealing to the robot's real-valued beliefs, like for example "as long as your (the robot's) confidence in whether the widget is flawed or not is below a threshold of 99%, (re-)inspect the widget." Note that in this plan the activation of the low-level inspect process is conditioned on the robot's beliefs *at execution time*.

In this chapter, we will present an approach to belief update which is based on the use of pGOLOG programs to represent (the internal state of) the low-level processes. Note that besides updating its beliefs concerning the state of the world in terms of fluents like PA or PR, the robot also has to update its beliefs concerning the state of execution of the low-level processes. For example, 15 seconds after activation of the *paint* process the robot should not only be aware of the fact that the widget is under-coated by now, but also that the process is

[1] The respective probabilities are 0.3*0.1=3% and 3/73=4.1%.

no longer in its initial state but only 15 seconds away from completion (recall that we assume that *paint* first undercoats the widget, and thereafter paints if; cf. Section 6.2.3). Roughly, then, we represent the robot's belief state by a set of *configurations* $\langle s', ll_{model} \rangle$ considered possible, where s' is a possible situation and ll_{model} a model of the low-level processes and their actual state of execution. Thereby, we use different pGOLOG programs to represent different states of execution of the low-level processes. For example, initially we characterize the *paint* process by *paintProc*, but 15 seconds after its activation we characterize it by what remains of *paintProc* after 15 seconds of execution. This approach can be formalized quite naturally in pGOLOG due to the fact that pGOLOG's semantics includes a specification of the continuation of a program.

Based on the characterization of the robot's epistemic state as a distribution over configurations, belief update works as follows. Initially, all low-level processes are wait-blocked. Whenever the high-level controller executes an action a, in the resulting situation $do(a, s)$ the possible configurations $\langle s', ll_{model} \rangle$ are replaced by configurations which results (roughly) from the execution of ll_{model} in s'. In particular, if a is a *send* action that activates a low-level process, then in the successor situation $do(a, s)$ all configurations considered possible will account for the fact that ll_{model} has become unblocked. For example, if a activates the paint process then in the following situations the configurations considered possible will account for the fact that *paintProc* is active and will subsequently cause *UC* and (possibly) *PA* to become true. Finally, whenever a *reply* action occurs, the high-level controller makes use of this action to sharpen its belief state: in every configuration considered possible it verifies whether this *reply* is "compatible" with the pGOLOG program ll_{model}, meaning that ll_{model} is about to execute this very *reply* action. Then, in the successor situation $do(a, s)$ it rules out all those configurations from the belief state which are not compatible with the observation. Intuitively, these configurations can be removed from the epistemic state because they do not correctly characterize the actual world.

This chapter is organized as follows: in the next section, we show how belief update can be realized in the pGOLOG framework. In Section 7.2 we make use of our framework to replicate an example from [BHL99]. Finally, we consider belief-based programs, and show how on-line execution and probabilistic projection can be interleaved.

7.1 On-line Execution and Belief Update

In this section, we will present our approach to belief update under probabilistic uncertainty, which is based on the use of probabilistic pGOLOG programs modeling (the state of execution of) the noisy low-level processes. However, before we can turn to the specification of belief update, we first have to consider on-line execution in pGOLOG.

7.1.1 On-Line Execution and On-line Execution Traces

To start with, similar to Chapter 5 we modify pGOLOG's semantics regarding *waitFor* actions to account for both projection and on-line execution. As in Section 5.1.3, we make use of the fluent $online(s)$ to distinguish between both modes of operation, and simply treat *waitFor* actions as tests during on-line execution. Formally, we replace the assertion for primitive actions in the first-order definition of *transPr* by a new axiom. As before, for readability we specify the new axiom for *transPr* in the **if** - **then** - **else** notation introduced in Section 6.1.1:

$$\begin{aligned}&transPr(\alpha, s, \delta, s') = \\ &\quad \textbf{if } [\neg online(s) \vee \forall \tau. \alpha \neq waitFor(\tau)] \wedge Poss(\alpha[s], s) \wedge \delta = \mathsf{nil} \wedge s' = do(\alpha[s], s) \textbf{ then } 1 \\ &\quad \textbf{else if } online(s) \wedge \exists \tau. \alpha = waitFor(\tau) \wedge \tau[s, start(s)] \wedge \delta = \mathsf{nil} \wedge s = s' \textbf{ then } 1 \textbf{ else } 0.\end{aligned}$$

As in cc-Golog, the semantics of *transPr* is as before if we are in projection mode: in situation s, there is a transition with probability 1 from the primitive or *waitFor* action α to a successor configuration $\langle \delta, s' \rangle$ if α is possible, $s' = do(\alpha[s], s)$ and $\delta = \mathsf{nil}$, and there is no other transition with positive weight. The same holds if we are in on-line execution mode and a is not a *waitFor* action. However, if we are in on-line execution mode and a is a $waitFor(\tau)$ action, then a is treated as test, meaning that there is a transition with positive weight if and only if the *t-form* τ is true at the beginning of the actual situation, that is if $\tau[s, start(s)]$ holds. If this is true, there is exactly one transition with positive weight, and the weight of this transition is 1. Furthermore, in the new configuration $s = s'$ and $\delta = \mathsf{nil}$.

Similarly, the second order definition of *transPr* is obtained by replacing the implication for primitive actions in Φ_{transPr} by the following two implications:

$$\begin{aligned}Poss(\alpha[s], s) \wedge [\neg online(s) \vee \forall \tau. \alpha \neq waitFor(\tau)] \supset t(\alpha, s, \mathsf{nil}, do(\alpha[s], s)) = 1 \\ online(s) \wedge \tau[s, start(s)] \supset t(waitFor(\tau), s, \mathsf{nil}, s) = 1.\end{aligned}$$

We remark that all propositions of the previous chapter regarding the properties of *transPr* and *transPr** continue to hold with respect to the new definition of *transPr*. The only difference is that in the proofs an additional base case has to be considered.

On-Line Execution Traces Now that we have adapted pGOLOG's semantics to account for on-line execution, let us consider the situations which can result from an actual on-line execution of a bGOLOG plan. As in Section 5.1 where we considered the on-line execution of cc-Golog plans, we assume that the execution system of the robot periodically provides the high-level controller with *ccUpdate*s that set the value of the continuous fluents to the latest estimates of the low-level processes. Besides updating continuous fluents, these actions also provide the correct current time; thus, we use the same successor state axiom for *start* as in Section 5.1.2. Finally, the low-level processes can execute *reply* actions to provide the high-level controller the answers of sensor processes. The following definition is completely analogous to the definition of on-line execution in cc-Golog (cf. Section 5.1.4).

Definition 19 *Let* $\text{AX}_{\text{pOnline}}$ *be the foundational axioms of the epistemic situation calculus together with the precondition axioms for waitFor and ccUpdate, the successor state axioms for start and online from Section 5.1.2, the (second-order) definitions of Final and transPr* from the previous chapter, the axioms needed for the encoding of programs as first-order terms, the axioms required for* t-form*'s, the set of axioms* AX_{arch} *from Section 4.3.1, and the new second-order definition of transPr. Furthermore, let* AX *be* $\text{AX}_{\text{pOnline}}$ *together with a situation calculus axiomatization of an application domain. Then an on-line execution with respect to* AX *of a program* σ_0 *in a situation* s_0 *is a sequence* $\sigma_0, s_0, ..., \sigma_n, s_n$ *such that for* $i = 0, ...n-1$*:*

1. $\text{AX} \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$*; or*
2. $\exists a, i, n.\ a = reply(i, n) \wedge \sigma_{i+1} = \sigma_i \wedge s_{i+1} = do(a, s_i)$*; or*
3. $\exists a, \vec{x}, t.\ a = ccUpdate(\vec{x}, t) \wedge [\forall \sigma', s'.\ transPr(\sigma_i, s_i, \sigma', s') > 0 \supset t \leq start(s')] \wedge \sigma_{i+1} = \sigma_i \wedge s_{i+1} = do(a, s_i)$.

We call an on-line execution completed *if* $\mathrm{AX} \models \mathit{Final}(\sigma_n, s_n)$. *Besides, we say that there is an on-line execution of* σ_0 *in* s_0 *that results in* (σ_n, s_n) *if and only if there is an on-line execution* $\sigma_0, s_0, ..., \sigma_n, s_n$ *of* σ_0 *in* s_0. *Finally, we say that a situation* s_n *is* a legal on-line execution trace of σ_0 in s_0 *if and only if there is a program* σ_n *such that there is an on-line execution of* σ_0 *in* s_0 *that results in* (σ_n, s_n).

The only difference between this definition and the definition of an on-line execution of a cc-Golog plans is that $\mathit{Trans}(\sigma, s, \delta, s')$ is replaced by $\mathit{transPr}(\sigma, s, \delta, s') > 0$. Note that as bGOLOG plans do not include prob instructions, a bGOLOG plan can always execute at most one action at any time (cf. Proposition 13).

As an example, let us reconsider the plan Π_{robby4} from Section 6.3.2:

$$\begin{aligned}\Pi_{robby4} \doteq [&\mathit{send}(\mathit{fork}, \mathit{inspect}), \mathsf{Bel}(\mathit{reg}(\mathit{inspect}) \neq \mathsf{nil}) = 1?,\\ &\mathit{send}(\mathit{fork}, \mathit{paint}), \mathsf{Bel}(\mathit{reg}(\mathit{painted}) = \top) = 1?,\\ &\mathsf{if}(\mathsf{Bel}(\mathit{reg}(\mathit{inspect}) = \mathit{OK}) = 1,\\ &\quad [\mathit{send}(\mathit{fork}, \mathit{ship}), \mathsf{Bel}(\mathit{reg}(\mathit{processed}) = \top) = 1?]\\ &\quad [\mathit{send}(\mathit{fork}, \mathit{reject}), \mathsf{Bel}(\mathit{reg}(\mathit{processed}) = \top) = 1?])].\end{aligned}$$

Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{pOnline}}$ together with an axiomatization of the ship/reject domain (cf. Section 6.3.2). The following situation is a legal on-line execution trace of Π_{robby4} in S_0 with respect to Γ, assuming that the execution system provides a *ccUpdate* action every 0.25 seconds (we only write the last argument (time) of *ccUpdate*):

$$\begin{aligned}S_1 \doteq \mathit{do}([&\mathit{send}(\mathit{fork}, \mathit{inspect}),\\ &\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{ccUpdate}(0.25), ..., \mathit{ccUpdate}(10.0),\\ &\mathit{reply}(\mathit{inspect}, \overline{\mathit{OK}}),\\ &\mathit{send}(\mathit{fork}, \mathit{paint}),\\ &\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{ccUpdate}(10.25), ..., \mathit{ccUpdate}(40.0), \mathit{reply}(\mathit{painted}, \top)\\ &\mathit{send}(\mathit{fork}, \mathit{reject}),\\ &\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{ccUpdate}(40.25), ..., \mathit{ccUpdate}(50.0), \mathit{reply}(\mathit{processed}, \top)], S_0).\end{aligned}$$

Proof: (Outline) A *send* action is always possible, so Π_{robby4} can execute $\mathit{send}(\mathit{fork}, \mathit{inspect})$. The low-level execution system answers by executing $\mathit{reply}(\mathit{fork}, \mathsf{nil})$. Thereafter, the remaining plan is blocked, waiting for $\mathit{Bel}(\mathit{reg}(\mathit{inspect}) \neq \mathsf{nil}) = 1$, that is for a $\mathit{reply}(\mathit{inspect}, \overline{\mathit{OK}})$ or $\mathit{reply}(\mathit{inspect}, \mathit{OK})$ action. So the low-level execution system can execute exogenous *ccUpdate* actions which cause time to advance (cf. the third condition in Definition 19). Then, the inspect process executes a $\mathit{reply}(\mathit{inspect}, \overline{\mathit{OK}})$ action, which unblocks the high-level plan. As a result, the plan activates the paint process and waits for $\mathit{Bel}(\mathit{reg}(\mathit{painted}) = \top) = 1$ to become true. Again, the low-level system replies by $\mathit{reply}(\mathit{fork}, \mathsf{nil})$, and executes a sequence of *ccUpdate* actions. Finally, the paint process signals completion by $\mathit{reply}(\mathit{painted}, \top)$. This reactivates the plan, which evaluates the conditional. The value of register *inspect* is $\overline{\mathit{OK}}$ because of the $\mathit{reply}(\mathit{inspect}, \overline{\mathit{OK}})$ answer of the *inspect* process, so the plan activates *reject*. Thereafter, it waits until the reject process executes $\mathit{reply}(\mathit{processed}, \top)$ and ends. □

Another example of an on-line execution trace is the following situation S_{UC} which results from the execution of the plan $[\mathit{send}(\mathit{fork}, \mathit{paint}), \mathsf{Bel}(\mathit{reg}(\mathit{painted}) = \top) = 1?]$. Note that in the following situation S_{UC} the execution of the high-level plan is not completed:

$$\begin{aligned}S_{UC} \doteq \mathit{do}([&\mathit{send}(\mathit{fork}, \mathit{paint}),\\ &\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{ccUpdate}(0.25), ..., \mathit{ccUpdate}(15.0)], S_0).\end{aligned}$$

We remark that our definition of an on-line execution does not ensure that an execution trace actually includes all the *reply* actions predicted by the pGOLOG model of the low-level processes. Nevertheless, the framework for belief update presented in this chapter relies on their occurrence; in particular, we assume that only such *reply* actions occur during on-line execution which are considered in the pGOLOG model of the low-level processes. So in a sense the definition of on-line execution is under-constrained. The argument is that pGOLOG programs like *kernelProc* are *designed* by the user as a *faithful model* of the actual low-level processes. That is, the user has to take care that if during on-line execution a particular *reply* action can occur, then there is a probabilistic branch in the pGOLOG model of the low-level processes involving this *reply* action.

Similarly, we assume that the difference $\Delta = t_{i+1} - t_i$ between two subsequent updates $ccUpdate(\vec{x}, t_i)$ and $ccUpdate(\vec{x}, t_{i+1})$ is smaller than the minimal delay between the execution time of any two actions of the pGOLOG models which have different execution time. Furthermore, we assume that if a *reply* is modeled to happen at time t, then during on-line execution the high-level controller's run-time system will generate a *ccUpdate* action causing *start* to advance to t before the actual *reply* action happens.

7.1.2 The Epistemic State as a Distribution over Configurations

As our notion of belief is defined in terms of the set of situations considered possible, in particular using the epistemic fluent p, it is obvious that one of the main tasks in specifying how a robot is to update its beliefs is the definition of an appropriate successor state axiom for the fluent p. However, specifying how the set of situations considered possible evolves is not sufficent. To see why, let us reconsider the situation S_{UC} where the robot has activated the *paint* process in the initial situation through $send(fork, paint)$, after which it has waited for 15 seconds. Recall that as described in Section 6.2.3 we assume that the paint process first undercoats the widget, and thereafter paints it, as specified by the following pGOLOG model:

$$\mathsf{proc}(paintProc, [clipBL, waitTime(10), \mathsf{if}(PR, setER, setUC), \\ waitTime(20), \mathsf{if}(PR, setER, \mathsf{prob}(0.95, setPA)), reply(painted, \top)]).$$

Intuitively, the robot's epistemic state in S_{UC} should reflect the fact that the activation of the low-level process *paint* has affected the truth value of *UC*, and, *additionally*, that unlike in S_0 the low-level process *paint* is now *active*, has already executed *setUC*, and is about to probably execute *setPA*. Thus, the *paint* process is no longer correctly characterized by *paintProc*, but instead by the remaining fragment of *paintProc* after 15 seconds have passed.

The example suggests that the appropriate pGOLOG model of the low-level processes is not the same for all situations, but depends on the history of actions. Thus we associate with every possible situation a specific pGOLOG model. Formally, we replace the binary fluent p by a ternary fluent *pll*.[2] $pll(s', ll', s)$ that can be read as "in situation s, the agent thinks that the world may be situation s' and that the low-level processes can be characterized by ll' with degree of likelihood $pll(s', ll', s)$." We require that initially every possible situation is associated with exactly one model of the low-level processes, and that every situation

[2] In earlier work [GL01a], we made use of two fluents $p(s', s)$ and $ll(s', s)$ to represent the epistemic state of the robot by a distribution over configurations. The fluent *pll* is related to p and ll as follows: $pll(s', ll', s) > 0 \equiv p(s', s) > 0 \wedge ll(s', s) = ll'$.

considered possible initially is initial. Formally:

$$[pll(s', ll', S_0) > 0 \wedge pll(s', ll^*, S_0) > 0] \supset [ll' = ll^* \wedge Init(s')]. \tag{7.1}$$

Here, $Init(s)$ is as defined in Section 3.3. Then, p can be understood as a restriction of pll ignoring the ll' argument, that is:

$$p(s', s) = p \equiv \exists ll'.pll(s', ll', s) = p. \tag{7.2}$$

In Section 7.1.5 (Proposition 22), we will show that p is well-defined given the above axiom.

As an example, the following axiom which replaces the initial state description from Section 6.2.1 says that the robot initially considers two situations possible, s_1 and s_2, with degree of likelihood 0.3 and 0.7, respectively, and that in both s_1 and s_2 the low-level processes are as described by *kernelProc*. These two configurations are the only configurations considered possible:

$$\begin{aligned} &\exists s_1, s_2.pll(s_1, kernelProc, S_0) = 0.3 \wedge p(s_2, kernelProc, S_0) = 0.7 \wedge \\ &\quad FL(s_1) \wedge BL(s_1) \wedge \neg PA(s_1) \wedge \neg PR(s_1) \wedge \neg ER(s_1) \wedge \\ &\quad \neg FL(s_2) \wedge \neg BL(s_2) \wedge \neg PA(s_2) \wedge \neg PR(s_2) \wedge \neg ER(s_2) \wedge \\ &\quad \forall s', ll'.\ pll(s', ll', S_0) > 0 \supset ll' = kernelProc \wedge \\ &\qquad [s' \neq s_1 \wedge s' \neq s_2] \supset pll(s', ll', S_0) = 0. \end{aligned} \tag{7.3}$$

7.1.3 Belief Update

Let us now turn to the specification of a successor state axiom for the epistemic fluent *pll*, stating how the robot's epistemic state evolves from a situation s to a successor situation $do(a, s)$. Our approach is based on the following assumptions: every answer of a sensor process is provided by means of an exogenous *reply* action, and the high-level controller is aware of all exogenous *reply* actions. On the other hand, the high-level controller is not aware of any other "action" executed by the low-level processes, like, for example, the *setPA* action which we used to model the effects of the low-level processes. In order to specify a successor state axioms for *pll*, then, we have to distinguish three cases: (i) a is an action executed by the high-level controller; (ii) a is a *ccUpdate* action; and (iii) a is a *reply* action performed by a sensor process. Note that these three cases correspond to the three cases in the definition of an online execution (cf. Definition 19).

Ordinary Actions Unlike *reply* actions which provide sensing information, actions executed by the high-level controller and *ccUpdate* actions do not provide any relevant informations about the actual state of the world, and thus cannot be used to sharpen the robot's beliefs. Therefore, we treat case (i) and (ii) similarly. Intuitively, all the robot can do if it executes an action a or if a *ccUpdate* action occurs is to update its beliefs, which are characterized by a set of configurations considered possible, by simulating how the possible configurations evolve from s to $do(a, s)$. Formally, if a is an action executed by the high-level controller or a *ccUpdate* action, then the configurations considered possible in s execute up to the point where one of the following conditions occur:

1. they are blocked;
2. or they are about to execute a *reply* action.

While the first condition is fairly obvious, the second condition reflects the intuition that the high-level controller is "aware" of all *reply* actions; as a is no *reply* action, the low-level processes cannot (yet) have executed a *reply* action.

We will now formalize the idea of executing a program σ in s until a configuration $\langle \delta, s' \rangle$ is reached where one of the above conditions is true. For this, we define the special function $transPr^{\lhd}(\sigma, s, \delta, s')$ which specifies the probability that a simulation of σ is s which does not involve the execution of a *reply* action results in $\langle \delta, s' \rangle$. In the following formulas, $isReply(a)$ is a shorthand for $\exists r, v.a = reply(r, v)$:

$$
\begin{array}{l}
transPr^{\lhd}(\sigma, s, \delta, s') = \\
\quad \textbf{if } transPr^{*}(\sigma, s, \delta, s') > 0 \wedge \\
\qquad \forall a^*, s^*.[s \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s'] \supset \neg isReply(a^*) \wedge \\
\qquad \forall \delta^*, s^*.transPr(\delta, s', \delta^*, s^*) > 0 \supset [\exists a^*.s^* = do(a^*, s') \wedge isReply(a^*)] \\
\quad \textbf{then } transPr^{*}(\sigma, s, \delta, s') \textbf{ else } 0.
\end{array}
$$

While the second line of the **if** condition verifies that no *reply* action was executed, the third line ensures that the configuration $\langle \delta, s' \rangle$ satisfies one of the above two conditions, meaning that the simulation has been pursued as far as possible.

***reply* actions** Now that we have formalized how the low-level processes evolve if a is an ordinary action, let us turn to the other case where a is a *reply* action. The reason that we distinguish *reply* actions from other actions is that *reply* actions provide *sensing information*, and can thus be used to sharpen the robot's beliefs. For example, if the robot observes a $reply(inspect, \overline{OK})$ action after activation of *inspect*, it can rule out those configurations from its belief state where $\neg FL$ holds, because the inspect process never erroneously reports $\overline{OK}$. In general, we use the observation of a *reply* action to *rule out* those configurations so far considered possible which are not about to execute this very *reply* action. Intuitively, this reflects the fact that if a configuration is not compatible with the *reply* action, it cannot be a correct characterization of the actual state of the world.

Before we formally define a successor state axiom for *pll*, we first define an auxiliary predicate *pll4*. Like *pll*, $pll4(s^*, ll^*, s, p)$ represents the probability p of a configuration $\langle s^*, ll^* \rangle$ in situation s. Based on *pll4*, we will define *pll* as:

$$S_0 \prec s \supset [pll(s^*, ll^*, s) = p \equiv pll4(s^*, ll^*, s, p)].$$

The reason why we do not directly define *pll* is that we first have to show that the probability p in $pll4(s^*, ll^*, s, p)$ is uniquely determined for every configuration $\langle s^*, ll^* \rangle$ and every situation s rooted in S_0. The following initial state axiom specifies that initially *pll* and *pll4* agree on the weight of the configurations considered possible:

$$pll4(s^*, ll^*, S_0, p) \equiv pll(s^*, ll^*, S_0) = p. \tag{7.4}$$

Finally, the following axiom specifies how *pll4* changes its value from s to $do(a, s)$:[3]

$$
\begin{array}{l}
S_0 \prec s \supset [pll4(s^*, ll^*, do(a, s), p) \equiv \\
\quad [\exists s', ll', s'', ll'', p', p^*.\ pll4(s', ll', s, p') \wedge p' > 0 \wedge
\end{array}
$$

[3] We remark that this definition is somewhat similar to a guarded successor state axiom [dGL99a]. However, it is not a guarded successor state axiom because the condition at the place of the "guard" mentions the situation term S_0.

$$\begin{aligned}
&\mathit{transPr}^{\lhd}(ll', s', ll'', s'') = p^* \wedge p^* > 0 \wedge s^* = \mathit{do}(a, s'') \wedge p = p' * p^* \wedge \\
&(\neg \mathit{isReply}(a) \wedge ll^* = ll'' \vee \mathit{isReply}(a) \wedge \mathit{transPr}(ll'', s'', ll^*, s^*) = 1)] \\
\vee p = 0 \wedge \neg[&\exists s', ll', s'', ll'', p'.\ \mathit{pll4}(s', ll', s, p') \wedge p' > 0 \wedge \mathit{transPr}^{\lhd}(ll', s', ll'', s'') > 0 \\
&\wedge s^* = \mathit{do}(a, s'') \wedge (\neg \mathit{isReply}(a) \wedge ll^* = ll'' \vee \\
&\qquad \mathit{isReply}(a) \wedge \mathit{transPr}(ll'', s'', ll^*, s^*) = 1)]].
\end{aligned}$$

If a is an ordinary action, all configurations $\langle s^*, ll^* \rangle$ considered possible (that is having a positive probability) in $\mathit{do}(a, s)$ are successors of configurations $\langle s', ll' \rangle$ considered possible in s. In particular, we require that s^* is of the form $s^* = \mathit{do}(a, s'')$ and that $\mathit{transPr}^{\lhd}(ll', s', ll^*, s'')$ is positive, meaning that $\langle s'', ll^* \rangle$ is obtained from $\langle s', ll' \rangle$ by a sequence of transition as specified by $\mathit{transPr}^{\lhd}$. On the other hand, if a is a *reply* action, then $\langle s^*, ll^* \rangle$ is only considered possible in $\mathit{do}(a, s)$ if there is an intermediate configuration $\langle s'', ll'' \rangle$ such that $\langle s'', ll'' \rangle$ is obtained from $\langle s', ll' \rangle$ by a sequence of transition as specified by $\mathit{transPr}^{\lhd}$, *and* $\mathit{transPr}(ll'', s'', ll^*, s^*) = 1$ with $s^* = \mathit{do}(a, s'')$. Thus, the case where a is a *reply* action is similar to the other case but additionally requires that the configuration $\langle s'', ll'' \rangle$ is "compatible" with the *reply* action, meaning that $\langle s'', ll'' \rangle$ is about to execute this very *reply* action.

Note that by this definition, a *reply* action a has the effect to update the probability of the configurations considered possible in a way similar to Bayesian conditioning (cf. [RN95]). In particular, if a configuration c has several different successors (as specified by $\mathit{transPr}^{\lhd}$), from which one successor with weight p^* involves the execution of the *reply* action a, then the (unnormalized) probability of the successor configuration of c in $\mathit{do}(a, s)$ is p^* times the probability of c in s.

7.1.4 Examples

Before we formally show some properties of *pll4*, in particular that p and *pll* can safely be defined in terms of *pll4*, we will present some examples to provide an intuition as to how *pll4* evolves as a result of the execution of actions. Let us first reconsider the example situation S_{UC} from Section 7.1.1, where the robot has activated the *paint* process in the initial situation after which it has waited for 15 seconds:

$$\begin{aligned}
S_{UC} \doteq \mathit{do}([&\mathit{send}(\mathit{fork}, \mathit{paint}), \\
&\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{ccUpdate}(0.25), \ldots \mathit{ccUpdate}(15.0)], S_0).
\end{aligned}$$

Recall that the *paint* process is modeled as follows:

$$\begin{aligned}
\mathsf{proc}(\mathit{paintProc}, [&\mathit{clipBL}, \mathit{waitTime}(10), \mathsf{if}(\mathit{PR}, \mathit{setER}, \mathit{setUC}), \\
&\mathit{waitTime}(20), \mathsf{if}(\mathit{PR}, \mathit{setER}, \mathsf{prob}(0.95, \mathit{setPA})), \mathit{reply}(\mathit{painted}, \top)]).
\end{aligned}$$

Then from our axiomatization of *pll4* together with the initial epistemic state specification (7.3) one can conclude the following:

$$\begin{aligned}
&\mathit{pll4}(s', ll', S_{UC}, p') \wedge p' > 0 \supset \\
&\quad \exists s_0, p_0 . p_0 > 0 \wedge \mathit{pll4}(s_0, \mathit{kernelProc}, S_0, p_0) \wedge \\
&\qquad s' = \mathit{do}([\mathit{send}(\mathit{fork}, \mathit{paint}), \mathit{reply}(\mathit{fork}, \mathit{nil}), \\
&\qquad\qquad \mathit{ccUpdate}(0.25), \ldots, \mathit{ccUpdate}(10), \mathit{setUC}, \\
&\qquad\qquad \mathit{ccUpdate}(10.25), \ldots, \mathit{ccUpdate}(15)], s_0') \wedge \\
&\qquad ll' = \mathsf{withPol}([[\mathsf{nil}, \mathit{waitFor}(\mathit{clock} \geq 30)], \\
&\qquad\qquad\qquad \mathsf{if}(\mathit{PR}, \mathit{setER}, \mathsf{prob}(0.95, \mathit{setPA})), \mathit{reply}(\mathit{painted}, \top)], \\
&\qquad\quad \mathit{kernelProc}).
\end{aligned}$$

That is, the situations considered possible in S_{UC} are successors of (one of the two) situations considered possible initially. In particular, they result from a situation s_0 considered possible initially by execution of a set of actions which includes *all* actions executed in S_{UC}. *In addition* to the actions present in the history of S_{UC}, this set of actions includes the action *setUC*. This action has been executed by the model of the low-level processes, namely by *paintProc*. Note that this action guarantees that in S_{UC} the robot believes (with probability 100%) that the widget is painted, because it is present in *all* situations considered possible in S_{UC}.

Next, let us consider the models of the low-level processes considered possible in S_{UC}. Unlike in the initial situation, the low-level processes are not represented by the program *kernelProc*. Instead, they are modeled by what remains of *kernelProc*, and in particular of *paintProc* after 15 seconds of execution. Thus, *pll4* accounts for the fact that the state of execution of the low-level processes has changed from S_0 to S_{UC}.

Similarly, if the robot waits for another 15 seconds, then in the resulting situation

$$S_{almostPA} \doteq do([send(fork, paint), \\ reply(fork, \mathsf{nil}), ccUpdate(0.25), ...ccUpdate(30.0)], S_0)$$

the model of the low-level processes becomes unblocked because the $waitFor(clock \geq 30)$ condition becomes true, that is

$$pll4(s', ll', S_{almostPA}, p') \wedge p' > 0 \supset \\ \exists s''.s' = do(ccUpdate(30), s'') \wedge clock[s', start(s')] = 30$$

and hence in situation

$$S_{PA} = do(reply(painted, \top), S_{almostPA})$$

the model of the low-level processes might have executed *setPA* (with probability 95%):

$$pll4(s', ll', S_{almostPA}, p') \wedge p' > 0 \supset \\ \exists s''.[s' = do([ccUpdate(30), setPA, reply(painted, \top)], s'') \vee \\ s' = do([ccUpdate(30), reply(painted, \top)], s'')] \wedge \\ ll' = \mathsf{withPol}(\mathsf{nil}, kernelProc).$$

In particular, it is possible to show that the sum of the weights of the situations s' which include a *setPA* action is 95%.

As another example, let us consider the situation

$$S_{inspect} \doteq do([send(fork, inspect), reply(fork, nil), ccUpdate(0.25), ...ccUpdate(10.0)], S_0).$$

where the robot has first activated *inspect*, and then has waited for 10 seconds. Recall that the inspect process was modelled as follows:

$$\mathsf{proc}(inspectProc, [waitTime(7), \\ \mathsf{if}(PR, setER, \\ \mathsf{if}(BL, [waitTime(3), \mathsf{prob}(0.9, reply(inspect, \overline{OK}), \\ reply(inspect, OK))], \\ [waitTime(3), reply(inspect, OK)]))]).$$

Then, one can deduce that in $S_{inspect}$ two situations are considered possible, that is:

$$
\begin{array}{l}
pll4(s', ll', S_{inspect}, p') \wedge p' > 0 \supset \\
\quad [\exists s_0.s' = do([send(fork, inspect), reply(fork, nil), ccUpdate(0.25), ..., ccUpdate(10.0)], s_0) \\
\qquad \wedge BL(s_0) \wedge FL(s_0) \wedge p' = 0.3 \\
\qquad \wedge ll' = \mathsf{withPol}([[\mathsf{nil}, waitFor(clock \geq 10)], \\
\qquad\qquad\qquad \mathsf{prob}(0.9, reply(inspect, \overline{OK}), reply(inspect, OK))], \\
\qquad\quad kernelProc) \vee \\
\quad \exists s_0.s' = do([send(fork, inspect), reply(fork, nil), ccUpdate(0.25), ..., ccUpdate(10.0)], s_0) \\
\qquad \wedge \neg BL(s_0) \wedge \neg FL(s_0) \wedge p' = 0.7 \\
\qquad \wedge ll' = \mathsf{withPol}([[\mathsf{nil}, waitFor(clock \geq 10)], reply(inspect, OK)] \\
\qquad\quad kernelProc)].
\end{array}
$$

Again, the situations considered possible in $S_{inspect}$ are obtained from situations considered possible initially. The associated model of the low-level processes has evolved beyond the conditional appealing to the truth value of BL; intuitively, the program in the first disjunct correspond to the situation where the widget is flawed, and the second to the situation where it is not flawed. Due to the fact that the starting time of $S_{inspect}$ is 10, they are both unblocked and about to execute a *reply* action.

Let us now consider the situation which results if the *inspect* process provides a $\overline{OK}$ answer, that is executes $reply(inspect, \overline{OK})$ in $S_{inspect}$, leading to

$$S_{\neg ok} \doteq do(reply(inspect, \overline{OK}), S_{inspect}).$$

The *reply* action results in a *smaller* set of configurations considered possible, reflecting the fact that the $reply(inspect, \overline{OK})$ action sharpens the robot's beliefs. In particular, by the definition of *pll4* only those configurations are considered possible in $S_{\neg ok}$ whose program component was about to execute the $reply(inspect, \overline{OK})$ action, and hence are compatible with the $reply(inspect, \overline{OK})$ action. All other configurations are removed from the robot's belief state as they do not correctly describe the actual state of the world. In particular, we can deduce:

$$
\begin{array}{l}
pll4(s', ll', S_{\neg ok}, p') \wedge p' > 0 \supset \\
\quad \exists s_0.[s' = do([send(fork, inspect), reply(fork, nil), \\
\qquad\qquad ccUpdate(0.25), ..., ccUpdate(10.0), tossHead, reply(inspect, \overline{OK})], s_0) \\
\qquad \wedge BL(s_0) \wedge FL(s_0).
\end{array}
$$

Note that this means that there is only *one* configuration which remains, and that FL holds in the situation component of this configuration. This reflects the fact that the robot's beliefs in the widget being flawed rise to 100%.

Let us now consider the situation which results if the *inspect* process provides an OK answer, leading to situation

$$S_{ok} \doteq do(reply(inspect, OK), S_{inspect}).$$

Here, we can deduce that two configurations remain possible, that is:

$$
\begin{array}{l}
pll4(s', ll', S_{inspect}, p') \wedge p' > 0 \supset \\
\quad [\exists s_0.s' = do([send(fork, inspect), reply(fork, nil), \\
\qquad\qquad ccUpdate(0.25), ..., ccUpdate(10.0), tossTail, reply(inspect, OK)], s_0)
\end{array}
$$

$$\begin{array}{l} \wedge BL(s_0) \wedge FL(s_0) \wedge p' = 0.3 * 0.1 \vee \\ \exists s_0.s' = do([send(fork, inspect), reply(fork, nil), \\ \qquad ccUpdate(0.25), ..., ccUpdate(10.0), reply(inspect, OK)], s_0) \wedge \\ \neg BL(s_0) \wedge \neg FL(s_0) \wedge p' = 0.7]. \end{array}$$

The two configurations correspond to the case where (a) the widget is flawed but the inspect process overlooks the blemish, and (b) the widget is not flawed. The respective (unnormalized) weights of the two configurations is 0.3×0.1 (widget flawed but *inspect* fails) and 0.7×1.0 (no flaw). This means that there is a normalized probability of 0.03/0.73 that the widget is flawed.

7.1.5 Formal Properties

In this subsection, we will show that we can safely define p and pll in terms of $pll4$. In particular, we will show that in every situation s rooted in S_0 every configuration $\langle s^*, ll^* \rangle$ is assigned exactly one probability by $pll4$, and that for every configuration $\langle s^*, ll^* \rangle$ with positive probability the situation component s^* uniquely determines the program component ll^*. Finally, we will show that in the resulting theory the fluent *reg* is directly observable.

To start with, we show some properties of $transPr^{\triangleleft}$. The following proposition shows that $transPr^{\triangleleft}$ only assigns a positive weight to *maximal* configurations, meaning that if $transPr^{\triangleleft}(\sigma, s, \delta, s') > 0$ then there is no successor configuration $\langle \delta^*, s^* \rangle$ of $\langle \delta, s' \rangle$ which also satisfies $transPr^{\triangleleft}(\sigma, s, \delta^*, s^*) > 0$. As before, to simplify the presentation of the proofs we use the same symbols to denote terms and elements of the domain of interpretation; the meaning will be clear from the context.

Proposition 20: *Let* Γ *be the set of axioms* $\mathrm{AX}_{\mathrm{pOnline}}$ *(cf. Definition 19 on page 135) together with the definition of* $transPr^{\triangleleft}$*. Then:*

$$\begin{array}{l} \Gamma \models [transPr^{\triangleleft}(\sigma, s, \delta', s') > 0 \wedge transPr^{\triangleleft}(\sigma, s, \delta'', s'') > 0] \supset \\ \qquad [s \sqsubseteq s' \wedge s \sqsubseteq s'' \wedge \neg(s' \sqsubset s'')] \end{array}$$

Proof: The first two conjuncts in the implication (that is $s \sqsubseteq s'$ and $s \sqsubseteq s''$) follow directly from Proposition 14 (page 114) and the fact that by definition $transPr^{\triangleleft}(\sigma, s, \delta, s') > 0$ implies $transPr^{*}(\sigma, s, \delta, s') > 0$. The last conjunct follows by contradiction. Let us assume there are situations s, s' and s'' and programs σ, δ' and δ'' satisfying the implication's left-hand side such that $s' \sqsubset s''$. Then, by the definition of $transPr^{\triangleleft}$, we get:

$$\begin{array}{l} transPr^{*}(\sigma, s, \delta', s') > 0 \wedge \\ \quad \forall \delta^*, s^*.transPr(\delta, s', \delta^*, s^*) > 0 \supset [\exists a^*.s^* = do(a^*, s') \wedge isReply(a^*)]. \end{array} \tag{7.5}$$

By the same definition we also get:

$$\begin{array}{l} transPr^{*}(\sigma, s, \delta'', s'') > 0 \wedge \\ \quad \forall a^*, s^*.[s \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s''] \supset \neg isReply(a^*). \end{array} \tag{7.6}$$

Let M be any model of Γ. Then by Proposition 15 (page 115) there is a sequence of configurations $\sigma_1, s_1, ..., \sigma_n, s_n$ such that $\sigma_1 = \sigma, s_1 = s, \sigma_n = \delta'', s_n = s''$ and $M \models transPr(\sigma_i, s_i, \sigma_{i+1}, s_{i+1}) > 0$ for $i = 1, ...n-1$, and furthermore there is a $j, 1 \leq j < n$ such that $\sigma_j = \delta', s_j = s'$. Thus, M entails $transPr(\sigma', s', \sigma_{j+1}, s_{j+1}) > 0$, and by (7.6)

$\neg\exists a^*, s^*.s_{j+1} = do(a^*, s^*) \wedge isReply(a^*)$. Contradiction with (7.5). □

The next proposition says that the simulation via $transPr^{\lhd}$ of a configuration $\langle \sigma, s\rangle$ never results in two successor configurations which have the same situation component but different program components.

Proposition 21: *Let Γ be as in the previous proposition. Then:*

$$\Gamma \models transPr^{\lhd}(\sigma, s, \delta, s') > 0 \wedge transPr^{\lhd}(\sigma, s, \delta', s') > 0 \supset \delta = \delta'.$$

Proof: By contradiction. The proof is somewhat similar to that of Proposition 16. First, note that by definition $transPr^{\lhd}(\sigma, s, \delta, s') > 0$ implies $transPr^{*}(\sigma, s, \delta, s') > 0$ and

$$[s \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s'] \supset \neg isReply(a^*) \tag{7.7}$$

and

$$transPr(\delta, s', \delta^*, s^*) > 0 \supset [\exists a^*.s^* = do(a^*, s') \wedge isReply(a^*)]. \tag{7.8}$$

Similarly for $transPr^{\lhd}(\sigma, s, \delta', s') > 0$. Assume that there is a model M of Γ and that there are configurations $\langle \sigma, s\rangle$, $\langle \delta, s'\rangle$ and $\langle \delta', s'\rangle$ such that $\delta \neq \delta'$ and $M \models transPr^{\lhd}(\sigma, s, \delta, s') > 0 \wedge transPr^{\lhd}(\sigma, s, \delta', s') > 0$. By Proposition 14, we get $s \sqsubseteq s'$. Let us first consider the case where $s' = s$. By Proposition 8 (page 100) we can conclude that there is a sequence of transitions with positive weight seq from $\langle \sigma, s\rangle$ to $\langle \delta, s'\rangle$ and another sequence seq' from $\langle \sigma, s\rangle$ to $\langle \delta', s'\rangle$. Because $s' = s$, these transitions do not involve the execution of any new action, and in particular they do not involve any *tossHead* or *tossTail* action. By Proposition 13 (page 112), then, we get that either seq is a subsequence of seq' or vice versa. Without loss of generality, we assume $\mathsf{seq} \subseteq \mathsf{seq}'$. Furthermore, $\mathsf{seq} \neq \mathsf{seq}'$ because of the assumption $\delta \neq \delta'$, hence there must be a transition with positive weight from $\langle \delta, s'\rangle$ to a successor configuration. Again, this transition cannot involve the execution of an action because $s' = s$. However, by (7.8) every transition from $\langle \delta, s'\rangle$ to a successor configuration involves the execution of a *reply* action. Contradiction.

Let us now turn to the case $s \sqsubset s'$. Again, by Proposition 8 there is a sequence of transitions with positive weight $\sigma_1, s_1, ..., \sigma_n, s_n$ from $\langle \sigma, s\rangle$ to $\langle \delta, s'\rangle$ and another sequence $\sigma'_1, s'_1, ..., \sigma'_m, s'_m$ from $\langle \sigma, s\rangle$ to $\langle \delta', s'\rangle$. The two sequences must differ because $\delta \neq \delta'$. Let us first consider the case where one sequence is a proper subsequence of the other. Without loss of generality, we assume $m < n$ and $\langle \sigma_i, s_i\rangle = \langle \sigma'_i, s'_i\rangle$ for $i = 1, ...m$. By (7.8) every transition from $\langle \sigma_m, s_m\rangle$ to a successor configuration involves the execution of a *reply* action, that is $\exists a^*.isReply(a^*) \wedge s_{m+1} = do(a^*, s_m)$. However, by (7.7) and Proposition 14 we get $transPr(\sigma_m, s_m, \sigma_{m+1}, s_{m+1}) > 0 \supset \neg\exists a^*, s^*.[isReply(a^*) \wedge s_{m+1} = do(a^*, s^*)]$. Contradiction.

So the two sequences must differ at some point. Let k be the smallest index such that $\sigma_k = \sigma'_k$ and $s_k = s'_k$ but $\sigma_{k+1} \neq \sigma'_{k+1}$ or $s_{k+1} \neq s'_{k+1}$. That is, $M \models transPr(\sigma_k, s_k, \sigma_{k+1}, s_{k+1}) > 0$, $M \models transPr(\sigma_k, s_k, \sigma'_{k+1}, s'_{k+1}) > 0$ and $\neg(\sigma_{k+1} = \sigma'_{k+1} \wedge s_{k+1} = s'_{k+1})$. By Proposition 13, we get $s_{k+1} = do(tossHead, s_k) \wedge s'_{k+1} = do(tossTail, s_k)$ or vice versa. Thus, by Proposition 14 we get $do(tossHead, s_k) \sqsubseteq s'$ and $do(tossTail, s_k) \sqsubseteq s'$. Contradiction with the unique names axioms for situations. □

Next, we show some important properties of *pll*4.

Proposition 22: *Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{pOnline}}$ together with the definitions of $transPr^{\lhd}$ and pll4 and the Axioms 7.4 and 7.1. Then:*

1. *For every situation s such that $S_0 \prec s$ there are no two situations s', s'' considered possible in s such that s'' can be obtained from s':*

$$\Gamma \models [S_0 \prec s \wedge pll4(s', ll', s, p') \wedge p' > 0 \wedge pll4(s'', ll'', s, p'') \wedge p'' > 0] \supset \\ [\neg(s' \sqsubset s'') \wedge \neg(s'' \sqsubset s')].$$

2. *For every situation s such that $S_0 \prec s$ the weight of a configuration $\langle s', ll' \rangle$ considered possible is unique:*

$$\Gamma \models [S_0 \sqsubset s \wedge pll4(s', ll', s, p) \wedge p > 0 \wedge pll4(s', ll', s, p') \wedge p' > 0] \supset p = p'.$$

3. *For every situation s such that $S_0 \prec s$ and every situation s', there is at most one configuration $\langle s', ll' \rangle$ with positive weight in s:*

$$\Gamma \models [S_0 \sqsubset s \wedge pll4(s', ll', s, p') \wedge p' > 0 \wedge pll4(s', ll'', s, p'') \wedge p'' > 0] \supset ll' = ll''.$$

Proof: We show all three assertions simultaneously by induction on the number of actions one has to execute to obtain s from S_0. That is, we use the induction principle from [Rei01] which says that to show that $\phi(s)$ holds in all situations rooted in the initial situation S_0 we only have to show that $\phi(S_0)$ and $[\phi(s) \wedge S_0 \sqsubseteq s] \supset \phi(do(a, s))$.

In S_0, (1) holds because of (7.4) and (7.1) ; similarly, (2) holds because of (7.4); and finally (3) follows from (7.1). In the induction step, we show the three assertions as follows:

1. By contradiction. Assume there is an interpretation of Γ in which there are two configurations $\langle s', ll' \rangle$ and $\langle s'', ll'' \rangle$ such that $s' \sqsubset s''$ and $\exists p' > 0 \wedge pll4(s', ll', do(a, s), p')$ and $\exists p' > 0 \wedge pll4(s'', ll'', do(a, s), p')$ holds. Then by the definition of $pll4$, there must be two configurations $\langle ll'_p, s'_p \rangle$ and $\langle s'_a, ll'_a \rangle$ such that (the interpretation entails) $\exists p'.pll4(s'_p, ll'_p, s, p') \wedge p' > 0$ and $transPr^{\triangleleft}(ll'_p, s'_p, ll'_a, s'_a) > 0$ and $s' = do(a, s'_a)$.

   Similarly, there must be two configurations $\langle ll''_p, s''_p \rangle$ and $\langle s''_a, ll''_a \rangle$ such that $\exists p'.p' > 0 \wedge pll4(s''_p, ll''_p, s, p')$ and $transPr^{\triangleleft}(ll''_p, s''_p, ll''_a, s''_a) > 0$ and $s'' = do(a, s''_a)$. By induction hypothesis (1), neither $s'_p \sqsubset s''_p$ nor vice versa. If $s'_p = s''_p$, then, by induction hypothesis (3), ll'_p and ll''_p must be the same; but then by Proposition 20 we get $\neg(s'_a \sqsubset s''_a)$, and finally $\neg(s' \sqsubset s'')$ because $s' = do(a, s'_a)$ and $s'' = do(a, s''_a)$. Contradiction.

   So s'_p and s''_p must be different. By definition, $transPr^{\triangleleft}(ll'_p, s'_p, ll'_a, s'_a) > 0$ implies $transPr^{*}(ll'_p, s'_p, ll'_a, s'_a) > 0$, so by Proposition 14 (page 114) we get $s'_p \sqsubseteq s'_a$ and $s''_p \sqsubseteq s''_a$. Furthermore, we get $s'_p \sqsubseteq s'$ and $s''_p \sqsubseteq s''$ because of $s' = do(a, s'_a)$ and $s'' = do(a, s''_a)$. However, by induction hypothesis (1) we also get $\neg(s'_p \sqsubset s''_p)$ and $\neg(s''_p \sqsubset s'_p)$, which together with $s'_p \neq s''_p$ means that s' and s'' must be different. Contradiction.

2. By contradiction. Assume there is an interpretation of Γ in which there is a configuration $\langle ll', s' \rangle$ which at the same time has two different positive weights p and p' in $do(a, s)$, that is $pll4(s', ll', do(a, s), p)$ and $pll4(s', ll', do(a, s), p')$ and $p \neq p' \wedge p, p' > 0$. Again, by the definition of $pll4$, there must be two configurations $\langle ll'_p, s'_p \rangle$ and $\langle s'_a, ll'_a \rangle$ and a positive probability p'_p such that $pll4(s'_p, ll'_p, s, p'_p)$ and $transPr^{\triangleleft}(ll'_p, s'_p, ll'_a, s'_a) > 0$ and $s' = do(a, s'_a)$ and $p = p'_p * transPr^{\triangleleft}(ll'_p, s'_p, ll'_a, s'_a)$. Similarly, there must be two other configurations $\langle ll''_p, s''_p \rangle$ and $\langle s''_a, ll''_a \rangle$ and a positive probability p''_p such that $pll4(s''_p, ll''_p, s, p''_p)$ and $transPr^{\triangleleft}(ll''_p, s''_p, ll''_a, s''_a) > 0$ and $s' = do(a, s''_a)$ and $p = p''_p * transPr^{\triangleleft}(ll''_p, s''_p, ll''_a, s''_a)$.

First, we show that $\langle ll'_p, s'_p\rangle$ must be the same as $\langle ll''_p, s''_p\rangle$. Assume that this is not true. Then by induction hypothesis (3) s'_p and s''_p must be different. By induction hypothesis (1), we get $\neg(s'_p \sqsubseteq s''_p)$ and $\neg(s''_p \sqsubseteq s'_p)$. By the fact that $transPr^{\lhd}(ll'_p, s'_p, ll'_a, s'_a) > 0$ implies $transPr^*(ll'_p, s'_p, ll'_a, s'_a) > 0$ and Proposition 14 we get $s'_p \sqsubseteq s'_a$ and $s''_p \sqsubseteq s''_a$. Furthermore, as $s' = do(a, s'_a)$ and $s' = do(a, s''_a)$ we get $s'_p \sqsubseteq s'$ and $s''_p \sqsubseteq s'$. Contradiction with $\neg(s'_p \sqsubseteq s''_p) \wedge \neg(s''_p \sqsubseteq s'_p)$.

So $\langle ll'_p, s'_p\rangle$ must be the same as $\langle ll''_p, s''_p\rangle$. From this together with induction hypothesis (2), it follows that $p'_p = p''_p$. As by assumption $p \neq p'$, this means that $transPr^{\lhd}(ll'_p, s'_p, ll'_a, s'_a)$ must differ from $transPr^{\lhd}(ll''_p, s''_p, ll''_a, s''_a)$. However, by $s' = do(a, s'_a)$ and $s' = do(a, s''_a)$ we get $s'_a = s''_a$. Thus, $transPr^{\lhd}(ll'_p, s'_p, ll'_a, s'_a)$ must be $\neq transPr^{\lhd}(ll'_p, s'_p, ll''_a, s'_a)$. Contradiction with Proposition 21.

3. By contradiction. Assume there is an interpretation of Γ in which there are two configurations $\langle ll', s'\rangle$ and $\langle ll'', s'\rangle$ such that $ll' \neq ll''$ and $\exists p'.p' > 0 \wedge pll4(s', ll', do(a, s), p')$ and $\exists p''.p'' > 0 \wedge pll4(s', ll'', do(a, s), p'')$. As before, by the definition of $pll4$ there must be two configurations $\langle ll'_p, s'_p\rangle$ and $\langle s'_a, ll'_a\rangle$ such that $\exists p'.p' > 0 \wedge pll4(s'_p, ll'_p, s, p'_p)$ and $transPr^{\lhd}(ll'_p, s'_p, ll'_a, s'_a) > 0$ and

$$s' = do(a, s'_a) \wedge [\neg isReply(a) \wedge ll' = ll'_a \vee isReply(a) \wedge transPr(ll'_a, s'_a, ll', s') = 1]. \tag{7.9}$$

and another two configurations $\langle ll''_p, s''_p\rangle$ and $\langle s''_a, ll''_a\rangle$ s.t. $\exists p'.p' > 0 \wedge pll4(s''_p, ll''_p, s, p''_p)$ and $transPr^{\lhd}(ll''_p, s''_p, ll''_a, s''_a) > 0$ and

$$s' = do(a, s''_a) \wedge [\neg isReply(a) \wedge ll'' = ll''_a \vee isReply(a) \wedge transPr(ll''_a, s''_a, ll'', s') = 1]. \tag{7.10}$$

As in the induction step for part (2), we can show that $\langle ll'_p, s'_p\rangle$ must be the same as $\langle ll''_p, s''_p\rangle$. Similarly, we can show that $\langle ll'_a, s'_a\rangle$ must be the same as $\langle ll''_a, s''_a\rangle$. This, together with (7.9) and (7.10) implies:

$$s' = do(a, s'_a) \wedge [\neg isReply(a) \wedge ll' = ll'_a \vee isReply(a) \wedge transPr(ll'_a, s'_a, ll', s') = 1] \tag{7.11}$$

$$s' = do(a, s'_a) \wedge [\neg isReply(a) \wedge ll'' = ll'_a \vee isReply(a) \wedge transPr(ll'_a, s'_a, ll'', s') = 1]. \tag{7.12}$$

Let us first consider the case where a is no *reply* action. Here, we get $ll' = ll'_a = ll''$, contradiction. Next, consider the case where a is a reply action. Here, (7.11) and (7.12) imply $transPr(ll'_a, s'_a, ll', do(a, s'_a)) = 1$ and $transPr(ll'_a, s'_a, ll'', do(a, s'_a)) = 1$. From Proposition 13 (page 112), then, we get $ll' = ll''$. Contradiction.

□

The above proposition tells us that we can safely define pll and p in terms of $pll4$. In particular, the second part of the above proposition tells us that for every situation s, $S_0 \prec s$, and every configuration $\langle s', ll'\rangle$, the "weight" p of $\langle s', ll'\rangle$ is uniquely determined by $pll4$. Hence, for every situation s which may result from the on-line execution of a legal bGOLOG program, we define pll as follows:

$$onlineExecTrace(s) \supset [pll(s', ll', s) = p \equiv pll4(s', ll', s, p)]. \tag{7.13}$$

Here, *onlineExecTrace*(s) is a shorthand for:

$$S_0 \preceq s \wedge \forall a, s^*.\ do(a, s^*) \sqsubseteq s \supset [\exists \vec{x}.a = send(\vec{x}) \vee a = reply(\vec{x}) \vee a = ccUpdate(\vec{x}),$$

meaning that s is required to be rooted in S_0 and may only include *send*, *reply* or *ccUpdate* actions. Note that these three possibilities correspond to the three conditions in Definition 19 along with the assumption that a high-level bGOLOG plan may only executes *send* actions (cf. Section 6.2.4).

The reason why we only define the value of *pll* in terms of *pll4* for situations s which satisfy *onlineExecTrace*(s) is that we want to correctly project bGOLOG plans appealing to the robot's beliefs in non-initial situations which result from on-line execution (e.g. to interleave on-line execution and probabilistic projection). Hence, we have to ensure that for every situation s' considered possible in an on-line execution trace s, the sets of situations considered possible in s' and s coincide. In particular, this means that *pll* is required to be Euclidean, similarly to Section 6.3.1:

$$[S_0 \preceq s \wedge pll(s', ll', s) > 0 \wedge pll(s'', ll'', s) = p] \supset pll(s'', ll'', s') = p. \tag{7.14}$$

However, the above axiom is not sufficient to guarantee that the sets of situations considered possible in an execution trace s and in a situation s' coincide (where s' is a situation considered possible in s). Additionally, we have to ensure that there is no *second* situation s_2 ($S_0 \prec s_2$ and $s \neq s_2$), such that s' is also considered possible in s_2 (because s_1 and s_2 might consider a different set of situations as possible). To avoid running into ambiguities, we require that all situations rooted in S_0 which do *not* result from an on-line execution have an "empty" epistemic state:

$$S_0 \prec s \wedge \neg onlineExecTrace(s) \supset \forall s', ll'.pll(s', ll', s) = 0. \tag{7.15}$$

The following proposition tells us that the two Axioms (7.13) and (7.15) ensure that for every situation s', there is at most one situation s rooted in S_0 such that s' is considered possible in s:

Proposition 23: *Let* Γ *be the set of axioms* $\mathrm{AX}_{\mathrm{pOnline}}$ *together with the definitions of transPr*$^{\triangleleft}$*, pll4, pll and the Axioms 7.4, 7.1, 7.13 and 7.15. Then:*

$$\Gamma \models S_0 \prec s_1 \wedge S_0 \prec s_2 \wedge \exists ll'.pll(s', ll', s_1) > 0 \wedge \exists ll''.pll(s', ll'', s_2) > 0 \supset s_1 = s_2.$$

Proof: We first show the following: for every two situations s_1 and s_2 such that $S_0 \preceq s_1$, $S_0 \preceq s_2$ and $s_1 \sqsubset s_2$, the following holds:

$$pll(s', ll', s_2) > 0 \supset [\exists s'_p, ll'_p.s'_p \sqsubset s' \wedge pll(s'_p, ll'_p, s_1) > 0].$$

This follows by induction on the number of actions that lead from s_1 to s_2, using the definition of *pll*.

Using this property of *pll*, we show the proposition by contradiction. Suppose there is a situation s' considered possible in two different situations s_1 and s_2 rooted in S_0. Axiom (7.15) implies that s_1 and s_2 are on-line execution traces. Furthermore, s_2 must be a successor of s_1, or vice versa. Else s_1 and s_2 would differ at some point in their history, meaning that there are two different *send*, *reply* or *ccUpdate* actions a_1 and a_2 such that at some point in the history, s_1 includes a_1 while s_2 includes a_2. By the definition of *pll4*, if s' is considered possible

in s_1 then every action in the history of s_1 also occurs in the history of s'. Furthermore, s' may not include additional *send*, *reply* or *ccUpdate* actions not present in the history of s_1, because by assumption *send* and *ccUpdate* actions never occur in the (pGOLOG) models of the low-level processes (cf. Sections 6.2.3 and 5.1.1) and the high-level controller is aware of all *reply* actions, meaning that the low-level processes may only cause a *reply* action a to occur in s' if a also occurs in s_1 (cf. Section 7.1.3). Similarly, every action in the history of s_2 must occur in the history of s', and no additional *send*, *reply* or *ccUpdate* action may occur in s'. However, by assumption a_1 and a_2 are different, a contradiction.

So let us consider the case where one situation s_i is a successor of the other situation. Without loss of generality, we assume $s_1 \sqsubset s_2$. Then, by the above implication there must be a situation s'_p considered possible in s_1 such that $s'_p \sqsubset s'$. Contradiction with Proposition 22, part 1. □

Thus, for every situation s' considered possible in a situation rooted in S_0, the set of configurations considered possible in s' corresponds to the set of configurations considered possible in the uniquely determined situation s in which s' is considered possible. Hence, the sets of configurations considered possible in s and in s' coincide.

Now that we have defined how *pll* obtains its value in different situations, let us turn to the epistemic fluent p. The third part of the Proposition 22 tells us that we can safely define $p(s', s)$ as follows:

$$p(s', s) = p \equiv \exists ll'.pll(s', ll', s) = p. \tag{7.16}$$

Now that we have provided axioms which define the value of p in every situation s rooted in S_0, we will show that the fluent *reg* is *directly observable* with respect to our axiomatization. We first show the following property of *transPr**:

Proposition 24: *If the program σ does not include the primitive action a, then for any sequence of transitions leading from $\langle \sigma, s\rangle$ to $\langle \delta, s'\rangle$, s' is obtained from s without the execution of a and δ does not include a. Formally, let $\mathrm{AX}_{\mathrm{pOnline}}$ be the set of axioms from Definition 19 and let notIncludes(σ, a) be an expression which is true if and only if the program σ does not include the primitive action a (see Appendix A.3.1 for the details). Then:*

$$\begin{aligned}\mathrm{AX}_{\mathrm{pOnline}} \models\ & notIncludes(\sigma, a) \wedge transPr^*(\sigma, s, \delta, s') > 0 \\ & \supset notIncludes(\delta, a) \wedge \forall a^*, s^*.[s \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s'] \supset a^* \neq a.\end{aligned}$$

Proof: (Outline) First, we prove

$$\mathrm{AX}_{\mathrm{pOnline}} \models transPr(\sigma, s, \delta, s') > 0 \supset [notIncludes(\sigma, a) \supset [notIncludes(\delta, a) \wedge s' \neq do(a, s)]].$$

Using Proposition 11, this follows from the fact that $notIncludes(\sigma, a) \supset [notIncludes(\delta, a) \wedge s' \neq do(a, s)]$ satisfies the set of implications Φ_{Prop} from Section 6.1.4. Next, using Proposition 8 (page 100), we show the thesis for *transPr** by induction on the length of the sequence of transitions with positive weight from $\langle \sigma, s\rangle$ to $\langle \delta, s'\rangle$. □

We are now ready for the following proposition, which shows that *reg* is *directly observable* with respect to our axiomatization.

Proposition 25: *Let $\mathrm{AX}_{\mathrm{BelUp}}$ be the set of axioms $\mathrm{AX}_{\mathrm{pOnline}}$ together with the definition of transPr$^{\triangleleft}$, the axioms (7.13), (7.15) (7.14) and (7.16), the axiom for pll4 from Section 7.1.3,*

the axiom $p(s', S_0) > 0 \supset reg(id, s') = reg(id, S_0)$ *from Section 6.2.2 specifying that the initial value of reg is reflected in all situations* s' *considered possible initially, and the assertion* $pll(s', ll', S_0) > 0 \supset notIncludes(ll', send(id, val))$ *stating that a* pGOLOG *model of the low-level processes may not include a send action. Then reg is* directly observable *with respect to* $\mathrm{AX}_{\mathrm{BelUp}}$.

Proof: We prove the proposition by showing the stronger property that in every situation s rooted in S_0 the following holds:

$$pll(s^*, ll^*, s) > 0 \supset [notIncludes(ll^*, send(id, val)) \wedge reg(id, s^*) = reg(id, s)]. \tag{7.17}$$

By the induction principle [Rei01], we only have to show $\phi(S_0)$ and $[\phi(s) \wedge S_0 \sqsubseteq s] \supset \phi(do(a, s))$, where ϕ refers to the above property. In S_0, the property holds trivially because $\mathrm{AX}_{\mathrm{BelUp}}$ includes the two assertions $p(s', S_0) > 0 \supset reg(id, s') = reg(id, S_0)$ and $pll(s', ll', S_0) > 0 \supset notIncludes(ll', send(id, val))$.

In the induction step, we make use of the fact that, by definition, $pll(s^*, ll^*, do(a, s)) > 0$ implies that there are s', ll', s'', ll'', t such that $pll(s', ll', s) > 0$ and $transPr^{\lhd}(ll', s', ll'', s'') > 0$ and

$$s^* = do(a, s'') \wedge (\neg isReply(a) \wedge ll^* = ll'' \vee isReply(a) \wedge transPr(ll'', s'', ll^*, s^*) = 1). \tag{7.18}$$

By the induction hypothesis, we get $notIncludes(ll', send(id, val))$. Then, from (7.18), the fact that by definition $transPr^{\lhd}(ll', s', ll'', s'') > 0$ implies $transPr^{*}(ll', s', ll'', s'') > 0$ and Proposition 24, we can conclude $notIncludes(ll^*, send(id, val))$. Thus, we have shown the induction step for the first conjunct of the right-hand side of implication (7.17).

It remains to be shown that $pll(s^*, ll^*, do(a, s)) > 0 \supset reg(id, s^*) = reg(id, do(a, s))$. Again, from $notIncludes(ll', send(id, val))$, the definition of $transPr^{\lhd}(ll', s', ll'', s'') > 0$ and Proposition 24, we can conclude

$$\forall a^*, s^*.[s' \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s''] \supset \forall id, val.a^* \neq send(id, val). \tag{7.19}$$

Furthermore, the definition of $transPr^{\lhd}(ll', s', ll'', s'') > 0$ implies

$$\forall a^*, s^*.[s' \sqsubset do(a^*, s^*) \wedge do(a^*, s^*) \sqsubseteq s''] \supset \forall id, val.a^* \neq reply(id, val). \tag{7.20}$$

Recall that the successor state axiom for *reg* has the following form:

$$\begin{aligned} &Poss(a, s) \supset [reg(id, do(a, s)) = val \equiv \\ &\quad a = send(id, val) \vee a = reply(id, val) \vee \\ &\quad reg(id, s) = val \wedge \neg(\exists r, v.a = send(r, v) \vee a = reply(r, v))]. \end{aligned}$$

Then, from (7.19) and (7.20) and the successor state axiom for *reg* we get $reg(id, s') = reg(id, s'')$. Furthermore, by induction hypothesis $reg(id, s') = reg(id, s)$. Finally, by (7.18) and the successor state axiom for *reg* we get $reg(id, s^*) = reg(id, do(a, s'')) = reg(id, do(a, s))$, which finishes the proof. □

In Section 6.2.3, we introduced the continuous fluent *clock* which intuitively represents the actual time. It is straightforward to show that if initially the robot has perfect information about the value of *clock* then in the resulting theory *clock* is directly observable.

Proposition 26: *Let* $\mathrm{AX}_{\mathrm{BelUp}}$ *be as in Proposition 25. Furthermore, let* Γ *be* $\mathrm{AX}_{\mathrm{BelUp}}$ *together with the following axioms:*

1. $Poss(a, s) \supset clock(do(a, s)) = clock(s)$, *and*
2. $p(s', S_0) > 0 \supset clock(s') = clock(S_0)$.

Then clock is directly observable *with respect to* Γ.

Proof: (Outline) We have to show $S_0 \preceq s \supset [pll(s^*, ll^*, s) > 0 \supset clock(s') = clock(S_0)]$. Again, the thesis is shown using the induction principle [Rei01]. In S_0 the thesis is fulfilled because of axiom (2). In the induction step, the thesis is fulfilled because of axiom (1). □

This finishes the discussion of the formal properties of our specification of belief update.

7.2 Belief Update at Work - BHL's 1-Dimensional Robot

In the previous section, we have specified how the robot's epistemic state is to be updated, and have shown important properties of our axiomatization. In this section, we will show how our approach can be used to replicate the 1-dimensional robot example considered by Bacchus, Halpern and Levesque in [BHL99]. Here, we are given a mobile robot moving along a straight line in a 1-dimensional world. Initially, the robot believes to be at position 10 with probability of 50%. The other positions considered possible are 8, 9, 11 and 12, and each of these four possibilities is considered to have probability 12.5%. The robot can estimate its position by means of a noisy low-level process *noisySensePos*. *noisySensePos* has a 50% probability to correctly report the robot's position, and else reports a value that deviates by exactly one from the correct position. The probabilities for over- respectively for underestimation are equal. Finally, the robot can move by means of the low-level process *noisyAdv*. *noisyAdv* is also subject to noise; there is a 50% chance that it will correctly move the robot by the distance specified, otherwise it will move the robot one unit more or less with equal probability.

7.2.1 Specification of the domain

Let us first formally specify the initial epistemic state of our 1-dimensional robot. We use the functional fluent *position* to denote the robot's position, and assume that *position* can only take integer values. Initially, five situations are considered possible, and the value of *position* in these situations ranges from 8 to 12. The weight of the situation fulfilling $position = 10$ is 1/2, and the weight of the other four situation is 1/8. The following axiom makes this precise:[4]

$$\begin{aligned}&\exists s_1, s_2, s_3, s_4, s_5\ \forall s. s \neq s_1 \wedge s \neq s_2 \wedge s \neq s_3 \wedge s \neq s_4 \wedge s \neq s_5 \supset p(s, S_0) = 0 \wedge \\ &\quad p(s_2, S_0) = 1/8 \wedge position(s_2) = 8 \wedge p(s_3, S_0) = 1/8 \wedge position(s_3) = 9 \wedge \\ &\quad p(s_4, S_0) = 1/8 \wedge position(s_4) = 11 \wedge p(s_5, S_0) = 1/8 \wedge position(s_5) = 12 \wedge \\ &\quad p(s_1, S_0) = 0.5 \wedge position(s_1) = 10.\end{aligned} \tag{7.21}$$

It is easy to verify that the above axiom implies that the robot's beliefs regarding *position* are distributed as specified in the example description. In particular, let Γ be the set of

[4] We remark that as our approach does not require normalized probabilities, we could as well assign weight 4 to the situation s' satisfying $position(s') = 10$ and weight 1 to the other possible situations.

axioms $\mathrm{AX}_{\mathrm{BelUp}}$ from Proposition 25 together with the definition of *Bel* and the above axiom specifying the robot's initial epistemic state. Then it is easy to verify that Γ entails the following:

$$Bel(position = l, S_0) = \begin{cases} 1/8 \text{ if } l = 8 \\ 1/8 \text{ if } l = 9 \\ 1/2 \text{ if } l = 10 \\ 1/8 \text{ if } l = 11 \\ 1/8 \text{ if } l = 12 \\ 0 \quad \text{else.} \end{cases}$$

Noisy Sense Position Next, let us characterize the position estimation process by the pGOLOG procedure *estimateProc*. After activation, *estimateProc* provides an estimate of the actual position through a *reply* action affecting *reg*(*posEstimate*). To model that the estimate is correct with probability 50% and else deviates by one unit from the correct position, we make use of two nested prob instructions. Note that throughout this section, we represent low-level processes as instantaneous, abstracting away from their temporal extent.

$$\begin{array}{l} \mathsf{proc}(estimateProc, \mathsf{prob}(0.5, reply(posEstimate, position), \\ \qquad\qquad \mathsf{prob}(0.5, reply(posEstimate, position + 1), \\ \qquad\qquad\qquad reply(posEstimate, position - 1)))) \end{array}$$

At this point, it seems appropriate to clarify the sort of the different terms in this program. *position* is a (reified) functional fluent that is evaluated at simulation time; actually, it is a shorthand for the term *position*(*now*). On the other hand, *reply*(*posEstimate*, *position*) is a primitive action, and its effect is to assign a new value to *reg*(*posEstimate*), namely the value of *position* in the actual situation. Note that unlike *reg*, *position* is not directly observable.

Noisy Advance Now that we have modelled the position estimation process by the probabilistic program *estimateProc*, let us turn to the low-level process *noisyAdv*. We model the advance process by the pGOLOG procedure *advanceProc*, which takes as argument the integer d. *advanceProc* affects the position of the robot through the primitive action *exactAdv* (see below). To model that there is only a 50% probability that *noisyAdv* will move the robot by exactly d units, we use two nested prob instructions.

$$\begin{array}{l} \mathsf{proc}(advanceProc(d), \mathsf{prob}(0.5, exactAdv(d), \\ \qquad\qquad \mathsf{prob}(0.5, exactAdv(d + 1), exactAdv(d - 1)))) \end{array}$$

The following successor state axiom specifies that *position* changes its value only as an effect of the execution of the primitive action *exactAdv*(d). We assume that *exactAdv*(d) is always possible.

$$\begin{array}{l} Poss(a, s) \supset [position(do(a, s)) = l \equiv \\ \quad \exists d.a = exactAdv(d) \wedge l = position(s) + d \vee \forall d.a \neq exactAdv(d) \wedge l = position(s)] \end{array}$$

The Execution System's Kernel Process Next, we have to characterize the behavior of the whole execution level by a single pGOLOG procedure. The following procedure *kernelBHL* is similar to the procedure *kernelProc* used in Section 6.2.3. However, unlike *kernelProc*, *kernelBHL* has to deal with low-level processes involving arguments, namely with *noisyAdv*. In

particular, if *reg*(*fork*) is assigned the value *advance*(*d*), where *d* is an integer, then *kernelBHL* must call *advanceProc* with the argument *d*. To this end, we make use of the functional fluent *getAdvArg* defined below.

$$\begin{array}{l}\mathsf{proc}(kernelBHL, [reg(fork) \neq nil?,\\ \quad \mathsf{if}(reg(fork) = sense,\\ \quad\quad [reply(fork, nil), \mathsf{withPol}(estimateProc, kernelBHL)],\\ \quad\quad \mathsf{if}(\exists d.reg(fork) = advance(d),\\ \quad\quad\quad \mathsf{withPol}(advanceProc(getAdvArg), [reply(fork, nil), kernelBHL]),\\ \quad\quad [reply(fork, nil), kernelBHL]))])\end{array}$$

The functional fluent *getAdvArg* can be used to determine the number of units to move because it is defined to have value *d* if *reg*(*fork*) = *advance*(*d*):

$$\begin{array}{l}getAdvArg(x, s) = d \equiv reg(fork, s) = advance(d) \vee\\ \qquad\qquad\qquad\qquad d = 0 \wedge \forall d'.\ reg(fork, s) \neq advance(d').\end{array}$$

Finally, we can characterize the robot's initial epistemic state regarding the low-level processes by the following axiom:

$$pll(s', ll', S_0) > 0 \supset ll' = kernelBHL. \tag{7.22}$$

7.2.2 Dealing with Noisy Sensors

We will now discuss how the robot's beliefs evolve as a result of a sequence of activations of *noisySensePos* and *noisyAdv*. First, suppose the robot activates the noisy position estimation process, for example because it executes the following plan:

$$[send(fork, sense), \mathsf{Bel}(reg(posEstimate) \neq \mathsf{nil}) = 1?].$$

Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{BelUp}}$ from Proposition 25 together with the definition of *Bel* and the axioms from Section 7.2.1, that is the two axioms (7.21) and (7.22), the successor state axiom for *position*, the definition of *getAdvArg*, and the action precondition axiom $Poss(exactAdv(x)) \equiv \textsc{True}$. Then, it is easy to verify that there is an (uncompleted) on-line execution trace of the above program in S_0 that results in the following situation:

$$do([send(fork, sense), reply(fork, \mathsf{nil})], S_0).$$

We remark that the above situation represents the state of the world immediately before *noisySensePos* provides its estimate. It is possible to show that in this situation the robot's belief state is composed of situations that result from the execution of *send*(*fork*, *sense*) and *reply*(*fork*, nil) in an initial situation:

$$\begin{array}{l}\Gamma \models p(s', do([send(fork, sense), reply(fork, \mathsf{nil})], S_0)) > 0 \equiv\\ \quad \exists s'_0.\ p(s'_0, S_0) > 0\ \wedge s' = do([send(fork, sense), reply(fork, \mathsf{nil})], s'_0).\end{array}$$

Intuitively, the *send*(*fork*, *sense*) action unblocks the model of the low-level processes, which then immediately executes *reply*(*fork*, nil). Next, let us consider the state of execution of the low-level processes associated with the possible situations. Here, we get the following:

$$\begin{array}{l}\Gamma \models pll(s', ll', do([send(fork, sense), reply(fork, \mathsf{nil})], S_0)) > 0 \supset\\ \quad ll' = [\mathsf{nil}, \mathsf{withPol}(estimateProc, kernelBHL)].\end{array}$$

That is, the low-level process *noisySensePos* has been activated. Next, let us consider the following situation where *noisySensePos* has provided an estimate of 11. As before, it is possible to show that the following situation is a legal on-line execution trace:

$$S_{r1} \doteq do([send(fork, sense), reply(fork, \mathsf{nil}), reply(posEstimate, 11)], S_0).$$

Here, we can deduce that only three situations remain in the updated belief state (recall that initially five situations have been considered possible):

$$\begin{aligned}\Gamma \models & p(s', S_{r1}) = p \wedge p > 0 \equiv \exists s'_0.\ p(s'_0, S_0) > 0 \wedge \\ & [s' = do([send(fork, sense), reply(fork, \mathsf{nil}), tossTail, tossTail, reply(posEstimate, 11)], s'_0) \\ & \quad \wedge p = 1/2 * 1/4 \wedge position(s'_0) = 10 \vee \\ & \ s' = do([send(fork, sense), reply(fork, \mathsf{nil}), tossHead, reply(posEstimate, 11)], s'_0) \\ & \quad \wedge p = 1/8 * 1/2 \wedge position(s'_0) = 11 \vee \\ & \ s' = do([send(fork, sense), reply(fork, \mathsf{nil}), tossTail, tossHead, reply(posEstimate, 11)], s'_0) \\ & \quad \wedge p = 1/8 * 1/4 \wedge position(s'_0) = 12].\end{aligned}$$

These situations correspond to a) the robot being at position 10 and the position estimation process reporting $position + 1$; b) the robot being at position 11 and *noisySensePos*' estimate being perfect; or c) the robot being at position 12 and the position estimation process reporting $position - 1$. The respective (unnormalized!) weights are 1/2*1/4, 1/8*1/2 and 1/8*1/4. Thus, the total weight of all situations considered possible is 7/32, and as a result in S_{r1} the robot's updated (normalized) beliefs in its position are distributed as follows:

$$Bel(position = l, S_{r1}) = \begin{cases} 4/7 \text{ if } l = 10 \\ 2/7 \text{ if } l = 11 \\ 1/7 \text{ if } l = 12 \\ 0 \quad \text{else.} \end{cases}$$

We will now consider how the belief changes after a *sequence* of activations of *noisySensePos*. Let us assume that the agent re-activates *noisySensePos* in Situation S_{r1}, for example because it executes the plan

$$\begin{aligned}& [send(fork, sense), \mathsf{Bel}(reg(posEstimate) \neq \mathsf{nil}) = 1?, \\ & \ send(posEstimate, \mathsf{nil}), send(fork, sense), \mathsf{Bel}(reg(posEstimate) \neq \mathsf{nil}) = 1?].\end{aligned}$$

Note the $send(posEstimate, \mathsf{nil})$ action which is used to reset the value of $reg(posEstimate)$ to nil. This is needed to ensure that the second test $\mathsf{Bel}(reg(posEstimate) \neq \mathsf{nil}) = 1?$ will cause the high-level controller to wait until the *inspect* process has provided a *second* estimate. Furthermore, let us assume that the position estimation process yields another estimate of 11, resulting in the following situation:

$$S_{r2} \doteq do([send(estimate, \mathsf{nil}), send(fork, sense), reply(fork, \mathsf{nil}), reply(posEstimate, 11)], S_{r1}).$$

Note that S_{r2} is a legal on-line execution trace of the above plan in S_0. It is possible to deduce that in this situation, the robot considers the following situations as possible:

$$\begin{aligned}\Gamma \models & p(s', S_{r2}) = p \wedge p > 0 \equiv \exists s'_0.\ p(s'_0, S_0) > 0 \wedge \\ & [s' = do([send(fork, sense), reply(fork, \mathsf{nil}), tossTail, tossTail, reply(posEstimate, 11), \\ & \qquad send(estimate, \mathsf{nil}), send(fork, sense), \\ & \qquad reply(fork, \mathsf{nil}), tossTail, tossTail, reply(posEstimate, 11)], s'_0)\end{aligned}$$

$$
\begin{aligned}
&\wedge p = 1/2 * 1/4 * 1/4 \wedge \mathit{position}(s'_0) = 10 \vee \\
s' = \mathit{do}([&\mathit{send}(\mathit{fork}, \mathit{sense}), \mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{tossHead}, \mathit{reply}(\mathit{posEstimate}, 11), \\
&\mathit{send}(\mathit{estimate}, \mathsf{nil}), \mathit{send}(\mathit{fork}, \mathit{sense}), \\
&\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{tossHead}, \mathit{reply}(\mathit{posEstimate}, 11)], s'_0) \\
&\wedge p = 1/8 * 1/2 * 1/2 \wedge \mathit{position}(s'_0) = 11 \vee \\
s' = \mathit{do}([&\mathit{send}(\mathit{fork}, \mathit{sense}), \mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{tossTail}, \mathit{tossHead}, \mathit{reply}(\mathit{posEstimate}, 11), \\
&\mathit{send}(\mathit{estimate}, \mathsf{nil}), \mathit{send}(\mathit{fork}, \mathit{sense}), \\
&\mathit{reply}(\mathit{fork}, \mathsf{nil}), \mathit{tossTail}, \mathit{tossHead}, \mathit{reply}(\mathit{posEstimate}, 11)], s'_0) \\
&\wedge p = 1/8 * 1/4 * 1/4 \wedge \mathit{position}(s'_0) = 12].
\end{aligned}
$$

Analogous to the situations considered possible in S_{r1}, the situations considered possible in S_{r2} thus correspond to a) the robot being at position 10 and the position estimation process *twice* reporting *position* + 1; b) the robot being at position 11 and the position estimation process *twice* reporting *position*; and c) the robot being at position 12 and the position estimation process *twice* reporting *position* − 1. As a result, in S_{r2} the robot's beliefs about its positions are distributed as follows:

$$
\mathit{Bel}(\mathit{position} = l, S_{r2}) = \begin{cases} 4/9 \text{ if } l = 10 \\ 4/9 \text{ if } l = 11 \\ 1/9 \text{ if } l = 12 \\ 0 \quad \text{else.} \end{cases}
$$

Finally, suppose the robot activates *noisySensePos* yet another time, and that the estimate turns out to be 11 again. Then in the resulting situation S_{r3} the robot's beliefs about *position* are distributed as follows:

$$
\mathit{Bel}(\mathit{position} = l, S_{r3}) = \begin{cases} 4/13 \text{ if } l = 10 \\ 8/13 \text{ if } l = 11 \\ 1/13 \text{ if } l = 12 \\ 0 \quad \text{else.} \end{cases}
$$

7.2.3 Dealing with Noisy Effectors

The above examples illustrate how the activation of the sensor process *noisySensePos* sharpens the robot's belief state. Let us now consider the effects of an activation of *noisyAdv*. Suppose the robot decides to advance by 1 unit in Situation S_{r3}, leading to the following situation:

$$
S_{r4} \doteq \mathit{do}([\mathit{send}(\mathit{fork}, \mathit{advance}(1)), \mathit{reply}(\mathit{fork}, \mathsf{nil})], S_{r3}).
$$

In this situation, the robot only considers situations possible which end with a sequence of actions corresponding to one of the three different probabilistic branches of *advanceProc*. Intuitively, this is because the $\mathit{send}(\mathit{fork}, \mathit{advance}(1))$ action has activated the noisy advance process. In fact, it is possible to show

$$
\begin{aligned}
\Gamma \models\ & p(s', S_{r4}) = p \wedge p > 0 \equiv [\exists s''. \\
& s' = \mathit{do}([\mathit{send}(\mathit{fork}, \mathit{advance}(1)), \mathit{tossHead}, \mathit{exactAdv}(1), \mathit{reply}(\mathit{fork}, \mathsf{nil})], s'') \\
& \quad \wedge p = 1/2 * p(s'', S_{r3}) \vee \\
& s' = \mathit{do}([\mathit{send}(\mathit{fork}, \mathit{advance}(1)), \mathit{tossTail}, \mathit{tossHead}, \mathit{exactAdv}(1+1), \mathit{reply}(\mathit{fork}, \mathsf{nil})], s'') \\
& \quad \wedge p = 1/4 * p(s'', S_{r3}) \vee \\
& s' = \mathit{do}([\mathit{send}(\mathit{fork}, \mathit{advance}(1)), \mathit{tossHead}, \mathit{tossTail}, \mathit{exactAdv}(1-1), \mathit{reply}(\mathit{fork}, \mathsf{nil})], s'') \\
& \quad \wedge p = 1/4 * p(s'', S_{r3})].
\end{aligned}
$$

We remark that in total, there are 9 situations considered possible, which correspond to the three different execution traces of *advanceProc* in the three situations considered possible in S_{r3}. Note that the weight of the situations including an *exactAdv*(1) action is doubled compared to the weight of the class of situations including an *exactAdv*(0) or an *exactAdv*(2) action. Altogether, this results in the following distribution over the robot's beliefs regarding *position*:

$$Bel(position = l, S_{r4}) = \begin{cases} 4/52 \text{ if } l = 10 \\ 16/52 \text{ if } l = 11 \\ 21/52 \text{ if } l = 12 \\ 10/52 \text{ if } l = 13 \\ 1/52 \text{ if } l = 14 \\ 0 \quad \text{else.} \end{cases}$$

The above example illustrates that the activation of the noisy advance process causes the robot's confidence in its position to decrease. Now suppose the robot attempts to move back to its previous position by activating *noisySensePos* with -1 as argument. It is possible to show that robot's beliefs in the resulting situation

$$S_{r5} \doteq do([send(fork, advance(-1)), reply(fork, \mathsf{nil})], S_{r4})$$

are distributed as follows:

$$Bel(position = l, S_{r5}) = \begin{cases} 4/208 \text{ if } l = 8 \\ 24/208 \text{ if } l = 9 \\ 57/208 \text{ if } l = 10 \\ 68/208 \text{ if } l = 11 \\ 42/208 \text{ if } l = 12 \\ 12/208 \text{ if } l = 13 \\ 1/208 \text{ if } l = 14 \\ 0 \quad \text{else.} \end{cases}$$

That is, the re-activation of the noisy advance process further decreases the robot's confidence in its position. As a final example, let us assume that the robot activates *noisySensePos* yet another time, and that once again the low-level process reports 11. In particular, let us consider the situation S_{r6} defined as follows:

$$do([send(posEstimate, \mathsf{nil}), send(fork, inspect), reply(fork, \mathsf{nil}), reply(posEstimate, 11)], S_{r5}).$$

Then, we can deduce that the new activation of the position estimation process has sharpened the robot's belief state, resulting in the following beliefs regarding *position*:

$$Bel(position = l, S_{r6}) = \begin{cases} 57/235 \text{ if } l = 10 \\ 136/235 \text{ if } l = 11 \\ 42/235 \text{ if } l = 12 \\ 0 \quad \text{else.} \end{cases}$$

Finally, we remark that the distributions we obtained using belief update in the pGOLOG framework coincide with the distributions obtained in [BHL99], Section 7.4, and correspond to what one would expect using Bayesian conditioning.

7.3 Belief-Based Programs and Probabilistic Projection Tests

In this section, we will investigate two applications of belief update. First, we will show how based on the specification of how the fluents *pll* and p evolve it becomes possible to execute belief-based plans that appeal to the robot's real-valued beliefs at execution time. Second, we will show how to interleave on-line execution and probabilistic projection, in analogy to the interleaving of on-line execution and time-bound projection in cc-Golog considered in Section 5.2.

7.3.1 Belief-Based Programs

The belief-based bGOLOG plans we have considered so far were only *pseudo-belief-based plans*, that is belief-based plans whose tests and conditionals made only appeal to the robot's beliefs concerning the directly observable fluent *reg*. The reason why we only considered pseudo-belief-based plans is that *reg* is directly observable, and thus that the robot's beliefs regarding *reg* can be determined without considering the epistemic fluent p. Note that as *reg* is directly observable, the robot's beliefs regarding the value of a register are always either 0 or 100%. Now that we have formally specified how the epistemic fluents p and *pll* evolve, we can consider unrestricted belief-based plans that appeal to arbitrary real-valued beliefs.

As an example, let us go back to the ship/reject domain (cf. Chapter 6) and assume that we want to specify that the robot is to correctly process the widget with probability 99%. For simplicity, we do not require the widget to be painted. This can be achieved by the following belief-based bGOLOG plan:

$$\begin{array}{l}\mathsf{proc}(\Pi_{loopInsp},\\ \quad[\mathsf{while}(\neg(\mathsf{Bel}(FL) \geq 0.99 \vee \mathsf{Bel}(\neg FL) \geq 0.99),\\ \qquad [send(inspect, \mathsf{nil}), send(fork, inspect), \mathsf{Bel}(reg(inspect) \neq \mathsf{nil}) = 1?]),\\ \quad \mathsf{if}(\mathsf{Bel}(FL) = 1, send(fork, reject), send(fork, ship)), \mathsf{Bel}(reg(processed) \neq \mathsf{nil}) = 1?]).\end{array}$$

The above plan specifies that the robot is to activate the *inspect* process until it is sufficiently confident about whether the widget is flawed or not. Note that the $send(inspect, \mathsf{nil})$ action in the body of the while-loop is necessary to guarantee that the test $\mathsf{Bel}(reg(inspect) \neq \mathsf{nil}) = 1?$ blocks the plan until *inspect* has completed execution even if *inspect* has already been activated before. Once the robot is sufficiently confident about the value of FL, the widget is shipped or rejected, depending on the robot's beliefs. Let us now consider some example on-line execution traces of $\Pi_{loopInsp}$. First, let us consider the following situation:

$$\begin{array}{l}S_1 \doteq do([send(inspect, \mathsf{nil}), send(fork, inspect),\\ \qquad reply(fork, \mathsf{nil}), ccUpdate(0.25), ...ccUpdate(10.0),\\ \qquad reply(inspect, \overline{OK}),\\ \qquad send(fork, reject),\\ \qquad reply(fork, \mathsf{nil}), ccUpdate(10.25), ...ccUpdate(20.0), reply(processed, \top)], S_0).\end{array}$$

Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{BelUp}}$ (cf. Proposition 25) together with the axioms from Section 6.3.2 modeling the ship/reject domain. Then it is not difficult to see that S_1 is a legal (completed) on-line execution trace of $\Pi_{loopInsp}$ with respect to Γ. Initially, $\Pi_{loopInsp}$ can cause two transition, involving the execution of $send(inspect, \mathsf{nil})$ and $send(fork, inspect)$. Note that initially the while-loop is not *Final* because the robot's intial beliefs in FL are 0.3. Thereafter, the belief-based plan becomes blocked, waiting for $reg(inspect)$ to get a non-nil

value. The next actions in S_1 are exogenous actions. The $reply(inspect, \overline{OK})$ action causes the robot's beliefs in FL to immediately rise to 100% (cf. Section 7.1.4). As a result, the belief-based plan becomes unblocked. The while-loop is *Final* because in the actual situation $Bel(FL)$ is 1, so the belief-based plan pursues the conditional, which results in the execution of $send(fork, reject)$. Therafter, it waits for the *reject* process to finish execution, which is signaled by the exogeneous $reply(processed, \top)$ action, and finishes execution. Thus S_1 corresponds to a *completed* on-line execution.

The on-line execution of $\Pi_{loopInsp}$ can also result in other execution traces, like for example:

$$\begin{aligned} S_2 \doteq do([&send(inspect, \mathsf{nil}), send(fork, inspect),\\ &reply(fork, \mathsf{nil}), ccUpdate(0.25), ...ccUpdate(10.0), reply(inspect, OK),\\ &send(inspect, \mathsf{nil}), send(fork, inspect),\\ &reply(fork, \mathsf{nil}), ccUpdate(10.25), ...ccUpdate(20.0), reply(inspect, OK),\\ &send(fork, reject),\\ &reply(fork, \mathsf{nil}), ccUpdate(20.25), ...ccUpdate(30.0), reply(processed, \top)], S_0). \end{aligned}$$

As discussed in Section 7.1.4, the observation of *one* OK answer causes the robot's belief in $\neg FL$ to rise to $70/73 = 0.7/(0.7 + 0.3 * 0.1)$. Similarly, the observation of two OKs cause the robot's belief in $\neg FL$ to rise to $0.7/(0.7 + 0.3 * 0.1 * 0.1)$, which is more than 0.99. Thus, after the second OK answer the while-loop becomes *Final* and $\Pi_{loopInsp}$ executes the conditional.

As another example, the following belief-based plan specifies that the robot is to activate the *inspect* process until it is sufficiently confident about whether the widget is flawed or not, then it is to activate the *paint* process until its belief in the widget being painted rises to 99%, and finally it is to process the widget:

$$\begin{aligned} &\mathsf{proc}(\Pi_{loopInsp\&Paint},\\ &\quad [\mathsf{while}(\neg(\mathsf{Bel}(FL) = 1 \vee \mathsf{Bel}(\neg FL) \geq 0.99),\\ &\qquad [send(inspect, \mathsf{nil}), send(fork, inspect), \mathsf{Bel}(reg(inspect) \neq \mathsf{nil}) = 1?]),\\ &\quad [\mathsf{while}(\mathsf{Bel}(PA) \leq 0.99,\\ &\qquad [send(painted, \mathsf{nil}), send(fork, paint), \mathsf{Bel}(reg(painted) \neq \mathsf{nil}) = 1?]),\\ &\quad \mathsf{if}(\mathsf{Bel}(FL) = 1, send(fork, reject), send(fork, ship)), \mathsf{Bel}(reg(processed) \neq \mathsf{nil}) = 1?]). \end{aligned}$$

As the examples illustrate, belief-based programs allow the programmer to provide domain dependent procedural knowledge in a natural way. We remark that our framework does not only allow the on-line execution of belief-based plans like $\Pi_{loopInsp}$, but also supports probabilistic projection of belief-based plans. In particular, from the set of axioms Γ it is possible to deduce:

$$PBel(PR \wedge \neg ER, S_0, \Pi_{loopInsp}, kernelBHL) = 99.7, \text{ and}$$
$$PBel(PA \wedge PR \wedge \neg ER, S_0, \Pi_{loopInsp\&Paint}, kernelBHL) = 99.45075.$$

Functional Fluents in Belief-Based Programs In Section 6.2.4, we required that a bGOLOG plan may not refer to functional fluents as arguments of primitive actions or procedure calls. Intuitively, the reason why we had to make this assumption is that bGOLOG plans may only appeal to the robot's beliefs but not to the actual value of fluents. In particular, this means that a bGOLOG plan may not appeal to the value of functional fluents. Thus, the

following pGOLOG program is not a legal high-level plan with respect to our formalization of BHL's 1-dimensional robot example:

$$[say(\text{“My position is:”}), say(position)].$$

While it is clear that the robot cannot refer to the value of *position* because it is uncertain about it, intuitively nothing prevents the robot from announcing the *estimate* of the actual position, namely the value of $reg(posEstimate)$ provided by the low-level process *noisySensePos*. Note that unlike in the case of *position*, the robot is certain about the value of $reg(posEstimate)$, meaning that it has a 100% belief.

To allow the robot to refer to functional fluents about which it is certain, we introduce the epistemic functional fluent $Kwhich(f)$. *Kwhich* takes as argument a functional fluent f. Intuitively, the value of $Kwhich(f)$ is v if the robot has a 100% evidence that the value of f is v; else $Kwhich(f) = \mathsf{nil}$. The following axiom makes this precise:

$$Kwhich(f, s) = v \equiv Bel(f = v, s) = 1 \vee \neg\exists v'.Bel(f = v', s) = 1 \wedge v = \mathsf{nil}.$$

Thus, *Kwhich* allows the robot to refer to the value of functional fluents about which it has 100% beliefs. Using *Kwhich*, it would be possible to specify that the robot is to announce the estimate provided by *noisySensePos*, for example using the following instructions:

$$[say(\text{“My position is:”}), say(Kwhich(reg(posEstimate)))].$$

As another application of *Kwhich*, suppose the 1-dimensional robot wants to get to position 0. Then, taking advantage of *Kwhich*, one could specify the following plan, telling the robot to first activate *noisySensePos*, then wait until it provides an estimate d, and finally activate *noisyAdv*, telling it to move back the robot by d units:

$$\begin{aligned}\Pi_{kwhich} \doteq [&send(fork, sense), \mathsf{Bel}(reg(posEstimate) = \mathsf{nil}) < 1?,\\ &send(fork, advance(-1 * Kwhich(reg(posEstimate)))), \mathsf{Bel}(position = 0) > 0?].\end{aligned}$$

Using probabilistic projection, one can deduce that this plan has a reasonable probability to result in the robot being at position 0. Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{BelUp}}$ together with the definition of *PBel*, the axioms (7.21) and (7.22) from Section 7.2.1 specifying the robot's initial epistemic state, the successor state axiom for *position*, the definition of *getAdvArg*, and the action precondition axiom $Poss(exactAdv(x)) \equiv \textsc{True}$. Then, it is possible to show:

$$\Gamma \models PBel(position = 0, S_0, \Pi_{kwhich}, kernelBHL) = 3/8.$$

That is, the plan has a probility of 37.5% to move the robot to position 0. We remark that our formalism also entails that the probability to end up at position 1 respectively -1 is 25%, and that the probability to end up at position 2 respectively -2 is 6.25%. In total, there are 45 possible execution traces.

Aside – waitFor in Belief-Based Programs In Section 6.2.4 we also required that a bGOLOG plan may not include *waitFor* actions. Intuitively, the reason why we had to make this assumption is that while bGOLOG plans may only appeal to the robot's beliefs, *waitFor* actions directly appeal to the value of continuous fluents, like for example *clock*, *battLevel* or *robotLoc*. In the remainder of this subsection, we will sketch some consideration as to how this restriction can be overcome.

The idea is that although a bGOLOG plan may not wait for the value of a continuous fluent to fulfill certain conditions, it may very well wait for the robot's continuously changing *beliefs* about the value of continuous fluents to fulfill a condition. To get a feel for this idea, let us reconsider the 1-dimensional robot example from Section 4.1.2, where a mobile robot is moving along a straight line (this is not the example from Bacchus, Halpern and Levesque which doesn't account for continuous change). The robot's location is represented by the fluent *robotLoc1d*. For simplicity, we will not consider any actions that affect the robot's position, nor a model of the low-level processes.

Let us assume that initially the robot knows that it is moving with velocity 1, but is unsure about its position at the beginning of S_0: there is a 30% chance that it is at position 9, and a 70% chance that it is at position 10. The following axiom makes this precise, specifying that initially the robot considers two situations possible, s_1 and s_2, with degree of likelihood 0.3 respectively 0.7. s_1 represents the situation where at the beginning of S_0 the robot is at position 9, and s_2 where it starts at position 10. Note that in both situations, the value of *robotLoc1d* is a continuous linear function of time (cf. Section 4.1.3 on page 58).

$$\begin{aligned}&\exists s_1, s_2 \forall s. s \neq s_1 \wedge s \neq s_2 \supset p(s, S_0) = 0 \wedge \\ &\quad p(s_1, S_0) = 0.3 \wedge p(s_2, S_0) = 0.7 \wedge \\ &\quad \mathit{robotLoc1d}(s_1) = \mathit{linear}(9, 1, \mathit{start}(s_1)) \wedge \\ &\quad \mathit{robotLoc1d}(s_2) = \mathit{linear}(10, 1, \mathit{start}(s_2))\end{aligned}$$

Next, let us assume that the robot's task is to execute the action *say*("I am at 20") as soon as it reaches position 20. Intuitively, the robot cannot execute $\mathit{waitFor}(\mathit{robotLoc1d} \geq 20)$ because it has only probabilistic beliefs about its position. However, it can wait until its *belief* that it has reached position 20 exceeds a certain threshold. For example, it can wait for its belief in being at a position ≥ 20 to exceed 50%. Intuitively, this should be the case after 10 seconds. Or, it can wait for its belief to be at a position ≥ 20 to exceed 99%, which should cause it to wait for 11 seconds.

The above example suggests that one possibility to relax the condition that no *waitFor* actions may occur in bGOLOG plans would be to allow the occurence of *waitFor* actions appealing to *epistemic* t-forms. Here, an *epistemic* t-form is an expression of the form $\mathsf{Bel}(\tau)\ op\ p$, where τ is an ordinary *t-form*, $op \in \{\geq, =\leq\}$, and p is a probability. An example is $\mathsf{Bel}(\mathit{robotLoc1d} \geq 20) \geq 0.5$. As with ordinary *t-form*s, one would evaluate an epistemic *t-form* at a situation s and time t. $[\mathsf{Bel}(\tau)\ op\ p][s, t]$ would then be defined as $\mathit{Bel}(\tau[s, t])\ op\ p$, where $\tau[s, t]$ is the expression used to evaluate ordinary *t-form*s (cf. Section 4.1.4 on page 58). For example, $[\mathsf{Bel}(\mathit{robotLoc1d} \geq 20) \geq 0.5][s, t]$ would become $\mathit{Bel}(\mathit{val}(\mathit{robotLoc1d}(s), t) \geq 20) \geq 0.5$. Using this approach, it would be possible to specify and project bGOLOG plans like the following:

$$[\mathit{waitFor}(\mathsf{Bel}(\mathit{robotLoc1d} \geq 20) \geq 0.5), \mathit{say}(\text{"I am at 20"})].$$

However, these ideas can only be considered as pre-considerations to a more general framework for dealing with probabilistic uncertainty and continuous change in pGOLOG. There are many reasons why the considerations presented above cannot be considered complete. For one, so far we assume that during on-line execution the high-level controller is provided with estimates of the value of all continuous fluents by means of *ccUpdate* actions. This means that during on-line execution the continuous fluents are directly observable. However, in general it seems desirable to consider both directly observable and non-observable continuous fluents. For example, unlike the voltage level of the robot's batteries, which arguably can be considered

as directly observable during on-line execution, the (continuous) position of a ball kicked by the robot can hardly be considered as directly observable.

For another, our successor state axiom for *pll* specifies that a configuration c considered possible is only removed from the epistemic state (without replacement by a successor configuration) if a *reply* action occurs that is not compatible with the program component of c. However, when dealing with continuous change it is appealing to also make use of *ccUpdate* actions to sharpen the robot's epistemic state. For example, in the mobile robot example considered in Chapter 4 and 5 one could imagine to remove a configuration c from the robot's epistemic state if the approximation of the trajectory yielded by c and the estimates provided by the *ccUpdate*s differ significantly. We leave this to future work.

7.3.2 Probabilistic Projection Tests

As elaborated in Section 5.2, it is often useful to interleave on-line execution and projection, for example in order to deliberate over possible sub-plans before committing to one of them. In this subsection, we will discuss the changes necessary to allow for probabilistic projection during on-line execution. In particular, we will introduce local lookahead constructs which allows probabilistic projection and projection of the expected utility under user control.

Before we can turn to the definition of the projection tests, we first have to adapt our definition of projected belief to our characterization of the robot's epistemic state as a distribution over *configurations*. Note that the *PBel* construct introduced in Section 6.3.1 only allowed the projection of a plan using the *same* model of the low-level processes in all situations considered possible. On the other hand, in our characterization of the robot's epistemic state by the fluent *pll* the pGOLOG program modeling (the state of execution of) the low-level processes may differ from situation to situation.

Similar to Section 6.3, in situation s the projected probability that a sentence ϕ will hold after the execution of a plan σ is determined by simulating σ in each situation s' considered possible in s, weighted by the likelihood of s' as specified by p. The possible effects of the low-level processes are taken into account by concurrently simulating the pGOLOG model ll' associated with s'. Formally, let ϕ be a formula whose only term of sort situation is the special situation term *now*, s a situation, and σ a bGOLOG plan. Then we redefine $PBel(\phi, \sigma, s)$, the projected belief that that ϕ holds after the execution of σ in situation s, as follows:

$$
\begin{aligned}
&PBel(\phi, \sigma, s) = p \equiv \\
&\quad \exists p_{unorm}, s_{offline}.s_{offline} = do(clipOnline, s) \wedge \\
&\quad p_{unorm} = \textstyle\sum_{\{s', ll', s^* | \phi[s^*]\}} pll(s', ll', s_{offline}) \times doPr(\mathsf{withPol}(ll', \sigma), s', s^*) \wedge \\
&\quad p = p_{unorm} / \textstyle\sum_{\{s'\}} p(s', s_{offline})
\end{aligned}
$$

$PBel(\phi, \sigma, s)$ is thus defined to be the weight of all paths that reach a final situation s^* where ϕ holds, starting from a possible configuration $\langle ll', s' \rangle$, weighted by the robot's belief in $\langle ll', s' \rangle$. Note the use of *clipOnline*, which switches to projection mode both in the actual situation and in the situations considered possible. The main difference between the definition in Section 6.3 and this one is that while the former made use of the same pGOLOG model ll_{model} in every situation s', here the pGOLOG model of the low-level processes may vary over the different situations s' considered possible.

Now that we have adapted our definition of projected belief, we can turn to the definition of programs which appeal to probabilistic projection tests. In particular, similar to Section 5.2.2 we introduce a term PBel which is a reified version of *PBel* with situation argument

suppressed. The idea is that PBel may be used within tests, conditionals and while-loops to allow probabilistic projection under user control. Similar to Section 5.2.2, in order to avoid running into self-referencing sentences like programs appealing to their own effects, we introduce a new sort of programs: $Prog_{bGologPT}$. programs of sort $Prog_{bGologPT}$ are like bGOLOG programs but additionally may appeal to PBel conditions. On the other hand, PBel conditions are restricted to ordinary bGOLOG programs (as second argument) in order to avoid the definition of self-referencing programs. See Appendix A.3.4 for an in-depth treatment of these issues.

Similarly, we redefine the expected utility operator EU from Section 6.3.3, adapting it to our characterization of an epistemic state as a distribution over configurations. Formally, $EU(\sigma, s)$, the expected utility of the plan σ in situation s, is defined as the weighted average of the utility U (cf. Section 6.3.3) of the possible execution traces resulting from the execution of σ in s:

$$EU(\sigma, s) = u \equiv u = \sum_v v \times PBel(U(now) = v, \sigma, s).$$

Similarly to PBel, we define a term EU which is a reified version of EU and which may be used within programs of sort $Prog_{bGologPT}$; cf. Appendix A.3.4 for the details. Note that the use of EU requires an axiomatization of the utility and reward functions U and *reward*, just as in Section 6.3.3. We remark that in Section 8.4.3, we will present an extended example illustrating the use of the EU operator.

7.3.3 Dealing with Continuous Fluents in Probabilistic Projection Tests

If we want to interleave on-line execution and probabilistic projection in domains involving continuous fluents, new issues arise. These have to do with the fact that during on-line execution the low-level system periodically provides the high-level controller with updates of the values of the continuous fluents by means of *ccUpdate* actions, and that these actions affect the value of the continuous fluents.

To get a sense of the problem, let us reconsider the mobile robot example from Chapter 4 and 5. Here, we made use of the continuous fluent *robotLoc* to represent the robot's position, and approximated the robot's trajectory by a piecewise linear function. Let us assume that the robot, standing in the entrance of a room, starts traveling to a certain location l within the room. In this situation, an appropriate model of the low-level navigation process would specify that *robotLoc* is characterized by a linear function heading towards l, and that the low-level navigation process can be characterized as waiting until l is reached. The situation is illustrated in Figure 7.1 (left).

In order to enable the projection of the remaining plan *at any time* during on-line execution, in particular immediately after a *ccUpdate* action, we have to ensure that after a *ccUpdate* the continuous fluent *robotLoc* has as value a *t-function* which heads *towards the destination* (else the model of the low-level navigation process will wait forever). However, according to our current approach discussed in Section 5.1.1, *ccUpdate* would simply assign *robotLoc* a *constant* function of time, meaning that the robot would be projected never to get to its destination. Note that as illustrated by Figure 7.1 (right), specifying that instead *ccUpdate* is to shift the *t-function*, sticking to its derivatives, is no solution either because the resulting approximated trajectory may never reach the destination.

In order to overcome this problem, we need a new successor state axiom for *robotLoc* which specifies that after a *ccUpdate* action *robotLoc* has as value a *t-function* which starts at

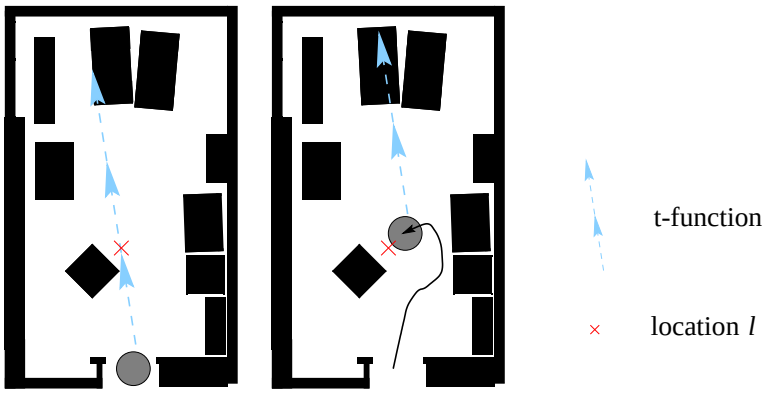


Figure 7.1: A shifted *t-function* missing the destination.

the robot's actual location but is heading towards the destination. For this purpose, we first define the new *t-function toCoords*. Similarly to *linear*, *toCoords* is a linear function of time. However, *toCoords* additionally provides information about the robot's destination. Formally, *toCoords* is a function with six argument: x_0, y_0, t_0, x_d, y_d, and v. The intuition is that at time t_0, *toCoords* has as value the tuple $\langle x_0, y_0\rangle$. Starting from there, $toCoords(x_0, y_0, t_0, x_d, y_d, v)$ moves linearly towards $\langle x_d, y_d\rangle$ with velocity v. The following axiom makes this precise:

$$\begin{aligned}
&val(toCoords(x_0, y_0, t_0, x_d, y_d, v), t) = \langle x, y\rangle \equiv \\
&\quad \neg[x_d = x_0 \wedge y_d = y_0] \wedge x = x_0 + (t - t_0) * v * (x_d - x_0)/\nu \wedge \\
&\qquad\qquad\qquad\qquad y = y_0 + (t - t_0) * v * (y_d - y_0)/\nu \vee \\
&\quad x_d = x_0 \wedge y_d = y_0 \wedge x = x_0 \wedge y = y_0.
\end{aligned}$$

Here, $\nu \doteq \sqrt{(x_d - x_0)^2 + (y_d - x_0)^2}$ is the total length of the arc from $\langle x_0, y_0\rangle$ to $\langle x_d, y_d\rangle$. This normalizing factor is needed in order to ensure that the total 2-dimensional velocity does not exceed v. We remark that the second disjunct of the above axiom ensures that the funtion is well defined even if $\langle x_0, y_0\rangle = \langle x_d, y_d\rangle$.

Using the new *t-function toCoords*, we can specify a new successor state axiom for *robotLoc* which ensures that if the value of *robotLoc* is a function of time heading towards a location l, then after a *ccUpdate robotLoc* is still heading towards l:

$$\begin{aligned}
&Poss(a, s) \supset [robotLoc(do(a, s)) = f \equiv \\
&\quad \exists t, x, y.\ t = start(do(a, s)) \wedge val(robotLoc(s), t) = \langle x, y\rangle \wedge v = 30cm/s \wedge \\
&\qquad [\exists x', y'. a = startGo(\langle x', y'\rangle) \wedge f = toCoords(x, y, t, x', y', v) \vee \\
&\qquad\ a = endGo \wedge f = constant(x, y) \vee \\
&\qquad\ \exists x_u, y_u, l_u, t_u. a = ccUpdate(x_u, y_u, l_u, t_u) \wedge \\
&\qquad\quad [\exists x', y', x'', y'', t', v'.\ robotLoc(s) = toCoords(x', y', t', x'', y'', v') \\
&\qquad\qquad \wedge f = toCoords(x_u, y_u, t_u, x'', y'', v) \vee \\
&\qquad\quad\ robotLoc(s) = constant(x, y) \wedge f = constant(x_u, y_u)] \vee \\
&\qquad \neg\exists x, y, l, t.[a = startGo(x, y) \vee a = endGo \vee a = ccUpdate(x, y, l, t)] \wedge \\
&\qquad\ f = robotLoc(s)]].
\end{aligned}$$

As in the old successor state axiom from Section 5.1.1, the variables x and y refer to the actual coordinates of the robot and t to the starting time of the new situation. After $startGo(\langle x', y'\rangle)$,

robotLoc has as value a *t-function toCoords* starting at the current position and moving toward $\langle x', y' \rangle$. After *endGo*, it is *constant*. If a is a $ccUpdate(x_u, y_u, l_u, t_u)$ action, things are somewhat more complicated: if the robot was travelling to location $\langle x'', y'' \rangle$, *robotLoc* is updated to a function *toCoords* whose value at the time where *ccUpdate* occurs is $\langle x_u, y_u \rangle$ and which is moving towards $\langle x'', y'' \rangle$; if the position was *constant*, it is updated to $constant(x_u, y_u)$. Finally, if a is neither a *startGo*, *endGo* nor a *ccUpdate* action, *robotLoc* remains unchanged. For simplicity, we model the robot as travelling at a constant speed of 30 cm/s.

Intuitively, by the new successor state axiom we *memorize* the robot's destination in the fluent *robotLoc*. To illustrate how this allows the projection of a plan dealing with continuous change immediately after a *ccUpdate* action, let us consider the following bGOLOG plan Π_{ccp}. The following examples are similar to the ones in Section 5.1.5, with the difference that here the robot's updated belief state keeps track of the internal state of exection of the low-level navigation process.

$$\Pi_{ccp} \doteq [send(destRoom, 6205), [reg(arrivedAt) = 6205]?]$$

We assume the following specification of the robot's initial epistemic state. For simplicity we only consider one situation possible:

$$\exists s^*, ll^*.ll^* = navProc \wedge robotLoc(s^*) = constant(p_0, q_0) \wedge battLevel(s^*) = 48 \wedge$$
$$[pll(s', ll', S_0) = p \wedge p > 0 \equiv s' = s^* \wedge ll' = ll^*].$$

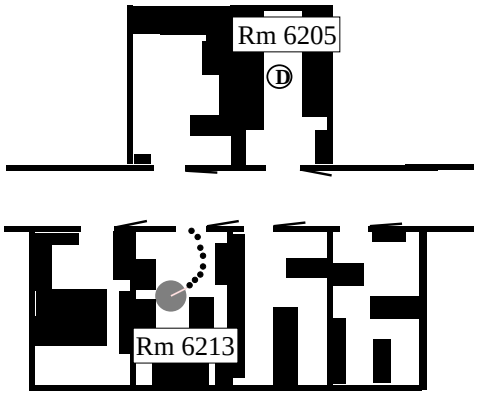


Figure 7.2: On-Line Execution Trace of Π_{ccp}

The above axiom specifies that initially the robot believes to be at position $\langle p_0, q_0 \rangle$, which we assume to be in Room 6213, and that the batteries voltage level is 48, which is sufficiently high to allow the robot to navigate. Furthermore, it specifies that initially the low-level navigation process (which in this example is the only low-level process) can be characterized by the procedure *navProc* described in Section 4.3.2. For convenience, we repeat the definition of *navProc* here:

$$\mathsf{proc}(gotoLoc(\langle x, y \rangle), [startGo(\langle x, y \rangle), waitFor(near(\langle x, y \rangle))])$$

$$\mathsf{proc}(travelTo(dest), \mathsf{conc}([\mathsf{if}(currentRoom \neq \text{HALLWAY},$$
$$[gotoLoc(exitOf(currentRoom)),$$
$$gotoLoc(entryOf(currentRoom))]),$$

$$\begin{array}{c} \mathit{gotoLoc}(\mathit{entryOf}(\mathit{dest})), \\ \mathit{gotoLoc}(\mathit{exitOf}(\mathit{dest})), \\ \mathit{gotoLoc}(\mathit{centreOf}(\mathit{dest})), \\ \mathit{endGo}, \mathit{reply}(\mathit{arrivedAt}, \mathit{dest})], \\ [(\mathit{reg}(\mathit{destRoom}) \neq \mathit{dest})?, \mathit{endGo}])) \end{array}$$

$$\begin{array}{l} \mathsf{proc}(\mathit{navProc}, \mathsf{forever}([(\mathit{reg}(\mathit{destRoom}) \neq \mathit{currentRoom} \wedge \mathit{reg}(\mathit{destRoom}) \neq \mathsf{nil})?, \\ \qquad\qquad \mathit{travelTo}(\mathit{reg}(\mathit{destRoom}))])). \end{array}$$

Let Γ be the set of axioms $\mathrm{AX}_{\mathrm{BelUp}}$ together with the new successor state axiom for *robotLoc*, the successor state axiom for *battLevel* and the action precondition axioms for *startGo* and *endGo* from Section 5.1.1, and the above axiom specifying the robot's initial epitemic state. Then, similar to Section 5.1.5, it is possible to show that the following situation S_{exec} visualized in Figure 7.2 is a legal on-line execution trace of the bGOLOG plan Π_{ccp} with respect to Γ. As usual, we assume that the navigation process provides a *ccUpdate* action every .25 seconds, and use p_i, q_i as appropriate $x-y$-coordinates along the path of the robot (we have left out the 3rd argument (voltage level) of *ccUpdate*):

$$\begin{array}{l} S_{exec} \doteq \mathit{do}([\mathit{send}(\mathit{destRoom}, 6205), \\ \qquad \mathit{ccUpdate}(p_1, q_1, 0.25), \mathit{ccUpdate}(p_2, q_2, 0.5), \\ \qquad \mathit{ccUpdate}(p_3, q_3, 0.75), \mathit{ccUpdate}(p_4, q_4, 1.0), \\ \qquad \mathit{ccUpdate}(p_5, q_5, 1.25), \mathit{ccUpdate}(p_6, q_6, 1.5), \\ \qquad \mathit{ccUpdate}(p_7, q_7, 1.75), \mathit{ccUpdate}(p_8, q_8, 2.0)], S_0). \end{array}$$

Similarly, it is not difficult to show that the following plan is what remains of Π_{ccp} in S_{exec}:

$$\Pi_{ccp}^{rem} \doteq [\mathsf{nil}, \mathit{reg}(\mathit{arrivedAt}) = 6205?].$$

Now let us take a look at the situations considered possible in S_{exec}. From Γ, it is possible to show that only one situation is considered possible in S_{exec}, and that this situation can be obtained from an initial situation s_0 by execution of the same actions as those which lead from S_0 to S_{exec} *plus* two additional *startGo* actions:

$$\begin{array}{l} \Gamma \models p(s', S_{exec}) > 0 \supset \exists s_0.p(s_0, S_0) > 0 \wedge \\ \quad s' = \mathit{do}([\mathit{send}(\mathit{destRoom}, 6205), \\ \qquad \mathit{startGo}(\mathit{exitOf}(6213)), \\ \qquad \mathit{ccUpdate}(p_1, q_1, 0.25), \mathit{ccUpdate}(p_2, q_2, 0.5), \\ \qquad \mathit{ccUpdate}(p_3, q_3, 0.75), \mathit{ccUpdate}(p_4, q_4, 1.0) \\ \qquad \mathit{ccUpdate}(p_5, q_5, 1.25), \mathit{ccUpdate}(p_6, q_6, 1.5), \\ \qquad \mathit{startGo}(\mathit{entryOf}(6213)), \\ \qquad \mathit{ccUpdate}(p_7, q_7, 1.75), \mathit{ccUpdate}(p_8, q_8, 2.0)], s_0). \end{array}$$

As in the examples in Section 7.1.4, the additional actions result from the fact that the model of the low-level processes (namely *navProc*) has become unblocked (because of the execution of $\mathit{send}(\mathit{destRoom}, 6205)$ by the high-level plan). In particular, *navProc* has first subsequently invoked *travelTo* and *gotoLoc*, which has finally executed $\mathit{startGo}(\mathit{exitOf}(6213))$. Then, it was blocked, waiting for the robot to reach a position near $\mathit{exitOf}(6213)$. We assume that $\langle p_6, q_6 \rangle$ is the first position which fulfills this condition. Thereafter, *navProc* has executed $\mathit{startGo}(\mathit{entryOf}(6213))$.

Next, let us consider the value of the continuous fluent *robotLoc* in the above situation s'. From Γ, we can deduce that $robotLoc(s')$ has as value a *t-function toCoords* starting at $\langle p_8, q_8\rangle$ and heading to $\langle x_1, y_1\rangle$, where x_1, y_1 stand for the $x-y$-coordinates of the intermeadiate position $entryOf(6213)$:

$$\Gamma \models p(s', S_{exec}) > 0 \supset$$
$$\exists t_8, v.start(s') = t_8 \wedge robotLoc(s') = toCoords(p_8, q_8, t_8, x_1, y_1, v).$$

Finally, let us consider the state of the pGOLOG model ll' associated with the situation s'. As one would expect, from Γ we can deduce that ll' corresponds to what remains of *navProc* after execution of $startGo(exitOf(6213))$ and $startGo(entryOf(6213))$:

$$\begin{aligned}\Gamma \models\ & pll(s', ll', S_{exec}) > 0 \supset \\ & ll' = [\mathsf{conc}([[\mathsf{nil}, waitFor(near(entryOf(6213)))], \\ & \qquad gotoLoc(entryOf(6205)), \\ & \qquad gotoLoc(exitOf(6205)), \\ & \qquad gotoLoc(centreOf(6205)), \\ & \qquad endGo, reply(arrivedAt, 6205)], \\ & \qquad [(reg(destRoom) \neq 6205)?, endGo]), \\ & \quad navProc].\end{aligned}$$

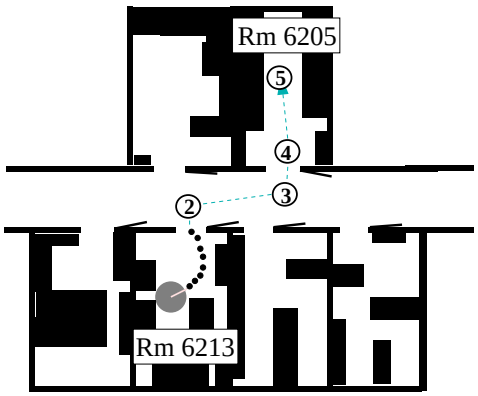


Figure 7.3: Projection during Execution

From the above, one can infer that it is possible to correctly project the remaining plan Π^{rem}_{ccp} in S_{exec}. To see why, observe that (a) $robotLoc(s')$ is heading towards intermediate position 2, that is towards $entryOf(6213)$ and (b) the pGOLOG model ll' accounts for the fact that the navigation process has already started execution and is now waiting to reach a position near $entryOf(6213)$. Figure 7.3 illustrates the simulation of the remaining plan in situation s' with ll' being used as model of the low-level navigation process.[5] Thus, the use of the new successor state axiom for *robotLoc* based on the *t-function toCoords* overcomes the problems

[5]We remark that the projection illustrated in Figure 7.3 slightly differs from the corresponding projection in Section 5.2.1. In particular, here the simulated execution trace includes a *waitFor* action at intermeadiate position l_2 which was not present in Figure 5.5 on page 91. This is because in Section 5.2.1 we did not memorize the state of execution of the low-level process, but instead used the whole program *navProc* as the model of the navigation process.

which arise when on-line execution and probabilistic projection is interleaved in the mobile robot domain which involves continuous change.

Finally, we remark that the approach presented in this subsection requires that the model of every low-level process is correct in the following sense: if it includes a *waitFor* action which is specified to always happen before a *reply* action, then during actual on-line execution the low-level process must be guaranteed to cause this *waitFor* action to become true before it provides the *reply* action. For example, if the model of the navigation process specifies that the navigation process first waits until $near(entryOf(6213))$ becomes true and only thereafter executes $reply(arrivedAt, dest)$, then during actual on-line execution the navigation process must really move the robot near the entry of Room 6213 before providing a $reply(arrivedAt, dest)$. If this is not guaranteed, the model of the low-level processes will lose synchronization with the actual state of the world, meaning that it will still be waiting for a *waitFor* condition like $near(entryOf(6213))$ to become true at the time where the *reply* action will occur. As a result, the model of the low-level process will not be compatible with that action (cf. Section 7.1.3). – Of course, if the user is not sure which path the navigation process will use, she can always specify a probabilistic model of the navigation process involving different probabilistic branches representing the different possible paths.

7.4 Discussion

In this chapter, we have specified how the robot is to update its probabilistic belief state during the on-line execution of high-level programs. Our approach accounts for noise, the temporal extent of low-level processes, and partial observability of the domain. It is based on the idea of characterizing the robot's epistemic state as a distribution over possible *configurations*, which consist of a situation modeling the possible state of the world, and a pGOLOG program modeling the low-level processes and their internal state of execution. Based on this characterization, belief update is realized, roughly, by simulating how the configurations considered possible evolve from s to $do(a, s)$. In doing so, *reply* actions which provide sensing information are used to sharpen the robot's epistemic state. In particular, if a *reply* action occurs then all configurations that did not predict the occurrence of this *reply* action are removed from the epistemic state. We have shown some important properties of our approach, and have proved the fluent *reg* to be directly observable with respect to our axiomatization.

Thereafter, we have illustrated our approach by showing that it can be used to replicate the 1-dimensional robot example considered in [BHL99]. Finally, we have shown how based on our approach it becomes possible to specify and execute belief-based plans, and how on-line execution and probabilisitic projection can be interleaved under user control. We have also elaborated on some issues which arise in the presence of continuous fluents. However, as discussed in the last paragraph of Section 7.3.1, there are still open issues regarding the seamless integration of continuous change, on-line execution and probabilistic belief update.

Many of the ideas in this paper rely on previous work by Bacchus, Halpern and Levesque [BHL95, BHL99]. In particular, we owe them the epistemic fluent p. Our approach and theirs differ in that we manage solely with the prob instruction, while they appeal to *nondeterministic instructions*, *action-likelihood* and *observation-indistinguishability axioms*. In particular, to represent that noisy sensors and effectors can have different possible outcomes, they make use of GOLOG's nondeterministic instructions $\sigma_1|\sigma_2$ and $\pi x.\sigma$. For example, to model a mobile robot moving along a straight line in a 1-dimensional world, they make use of the primitive

action $exactAdv(x, y)$, and of the nondeterministic action

$$noisyAdv(x) \doteq \pi y.exactAdv(x, y).$$

Here, $noisyAdv(x)$ represents an action directly executable by the robot, namely trying to move x, and $exactAdv(x, y)$ stands for the robot trying to move x but actually moving y. The idea is that the robot cannot directly execute the primitive action $exactAdv(x, y)$, but instead can only execute the nondeterministic action $noisyAdv(x)$. To ensure that the difference between the nominal value x and the actual value y is bound, they make use of appropriate precondition axioms, like for example:

$$Poss(exactAdv(x, y), s) \equiv |x - y| \leq b.$$

In order to represent the agent's limited perceptual capabilities, they introduce *observation-indistinguishability axioms* (OI-axioms) which are used to specifies that (in a certain situation) the agent is unable to discriminate a set of actions a' from the action a. For example, they specify that after execution of a concrete $exactAdv(x, y)$, the agent does not know which $exactAdv(x, y')$ was executed by the following observation-indistinguishability axiom:

$$OI(exactAdv(x, y), a', s) \equiv \exists y'.a' = exactAdv(x, y').$$

Additionally, they use action-likelihood axioms (l-axioms) to specify the likelihood of an action a in situation s. For example, to model that the agent believes that there is a 50% chance that $noisyAdv(x)$ will move the robot by a value y which deviates by one unit from the intended distance x they use the following axiom:

$$l(exactAdv(x, y), s) = \textbf{ if } x = y \textbf{ then } 0.5 \textbf{ else } \textbf{ if } |x - y| = 1 \textbf{ then } 0.25 \textbf{ else } 0.$$

Noisy sensors can be represented similarly. One effect of the different characterization of noisy low-level processes in their approach and in ours is that in the two approaches the situations s' considered possible in a situation s have a different relation to the actual history of actions in s. In their approach, the situations s' considered possible in s may only differ from the actual history in that some actions may have been replaced by "indistinguishable" actions. On the other hand, in our approach the situations considered possible may include *additional* actions besides those present in the actual history. Although their approach results in a simpler successor state axiom for p, this comes at the cost that it is not clear how to project a plan within their framework, which is straightforward in ours. In particular, our approach allows for probabilistic projection of programs appealing to projection tests and to the robot's real-valued beliefs at execution time.

As for the decision-theoretic DTGolog [BRST00, Sou01] whose semantics is based on the projection of the expected utility of a plan, it assumes full observability of the domain. On the other hand, DTGolog provides nondeterministic instructions, which – once again – are problematic in our approach. In particular, many of the properties shown in Section 7.1.5 which are necessary to guarantee that the epistemic fluent *pll* is well-defined would not hold if we would consider nondeterministic instructions.

Chapter 8

Implementation and Experimentation

In this chapter, we will sketch PROLOG implementations of interpreters for cc-Golog and pGOLOG. Just as in the case of ConGolog, although in the definition of the semantics we resorted to first- and second-order logic, it is relatively simple to provide a prototype implementation in PROLOG [dGLL00]. However, the implementation differs from the formal specification in that it makes the usual *closed world assumption* on the initial database. Furthermore, while the definition of pGOLOG makes use of second-order axioms to define summation, our implementation makes use of PROLOG set-predicates like `findall`. Finally, the implementation computes the transitive closure using *negation as failure* [Cla78], and makes use of the logic programming library clp(q,r) [Hol95] to deal with linear constraints. Note that this is a limitation of this particular implementation, not the theory.[1]

8.1 A cc-Golog Interpreter in PROLOG

We begin with the presentation of a cc-Golog interpreter in PROLOG. Unlike the ConGolog interpreter presented in [dGLL00], our interpreter does not only allow the use of relational fluents, but also of (non-continuous) functional fluents and, of course, of continuous fluents. The need for functional fluents, which were not present in the ConGolog implementation, stems from the fact that cc-Golog does not provide nondeterministic instructions.[2]

8.1.1 Legal Domain Specifications

Our implementation assumes that a domain specification consists of the follow parts:

[1] See [dGLL00] for an in-depth treatment of the relation between the second order axiomatization of ConGolog and a PROLOG implementation.

[2] In particular, ConGologs nondeterministic pick instruction $\Pi x.\sigma$ allows the introduction of a variable x, which can be bound at execution time and thereby allows the simulation of functional fluents. To see how the use of functional fluents can be overcome by using Π instructions, let us consider an example taken from [LRL+97], where a GOLOG program is used to control a simple elevator. In this example, the elevator controller has to serve the nearest floor as long as there is a floor which has to be served. Using a *functional* fluent $nextFloor(s)$, the task of serving the nearest floor can be specified as $serve(nextFloor)$, where $nextFloor(s) = n$ holds if and only if n is the nearest floor that has to be served in situation s. On the other hand, the same behavior can be realized by the nondeterministic GOLOG program $\Pi\ n.\ [nextFloor(n)?,serve(n)]$ which uses the *relational* fluent $nextFloor(n, s)$ which is true if and only if $nextFloor(s) = n$.

- A collection of clauses defining the predicate `poss(A,S)`, which specifies under which preconditions action `A` is executable in situation `S`.
- A collection of clauses defining the predicate `holds(F,s0)`, which specifies which fluents `F` hold in the initial situation `s0`.
- A collection of clauses defining the predicate `holds(F,do(A,S))`, which specifies the successor state axiom for every fluent `F`.
- A collection of clauses defining the predicate `hasValCF(CF,TFunc,s0)`, which specifies that the initial value of the continuous fluent `CF` is the *t-function* `TFunc`.
- A collection of clauses defining the predicate `hasValCF(CF,TFunc,do(A,S))`, which specifies the successor state axiom of every continuous fluent `CF`.
- A collection of clauses defining the predicate `isFF(FF)`, which is used to declare every functional fluent `FF`.
- A collection of clauses defining the predicate `hasValFF(FF,V,s0)`. `hasValFF/3` specifies, for each functional fluent `FF`, its initial value `V`.
- A collection of clauses defining the predicate `hasValFF(FF,V,do(A,S))`, which specifies the successor state axiom of every functional fluent `FF`.
- A collection of clauses defining the predicate `proc(procName,[`x_1`,...`x_n`],Body)`, used to define all cc-Golog procedures. In such clauses, the formal parameters of the procedure are represented by the *lower-case* PROLOG terms `x`$_1$`,...x`$_n$. Our implementation does not consider nested procedures.
- A collection of clauses defining the predicate `val(TFunc,V,T)`, which specifies the value `V` of the *t-function* `TFunc` at a given time `T`. These clauses implement the function *val* introduced in Section 4.1.3.

8.1.2 Legal cc-Golog Programs

The following PROLOG terms are considered as legal representations of cc-Golog programs:

- `nil`, the empty program;
- `a`, where `a` is the name of a user-declared primitive action or defined procedure;
- `p?`, where `p` is a condition as described below;
- the sequence `[`a_1`,...,`a_n`]`, where every a_i is a cc-Golog program;
- the conditional `if(p,`a_1`,`a_2`)`, where `p` is a condition and a_1 and a_2 cc-Golog programs;
- the while loop `while(p,a)`, where `p` is a condition and `a` a cc-Golog program;
- the construct `withCtrl(p,a)`, where `p` is a condition and `a` a cc-Golog program;
- the concurrent construct `conc(`a_1`,`a_2`)`, where a_1 and a_2 cc-Golog programs;
- and finally `waitFor(tf)`, where `tf` is a *t-form* as described below.

The argument of a primitive action or a procedure call is either an ordinary object, a functional fluent (as defined by `isFF/1`), or an expression `f1+f2`, `f1-f2`, `f1*f2`, `f1/f2`, where `f1` and `f2` are PROLOG numbers or numerical functional fluents. A condition `p` is either a PROLOG-term representing an atomic formula with the situation arguments suppressed, or an expression `and(p1,p2)`, `or(p1,p2)`, `not(p)` or `exists(v,p)`, with the obvious intended meaning. In the last case, `v` is a PROLOG constant, standing for a logical variable, and `p` a condition using `v`. A *t-form* `tf` is either a PROLOG-term `cf >= x` or `cf =< x`, where `cf` is a continuous fluent and `x` a number, or an expression `and(tf1,tf2)` or `or(tf1,tf2)`, with the obvious intended meaning. Additionally, our implementation supports the definition of user-defined *t-form*s. We assume a clause `holdsTFormMacro(TForm,S,T)` for each user-defined *t-form*, stating whether `TForm` holds at time `T` in situation `S`.

8.1.3 Dealing with Temporal Constraints

In order to deal with temporal constraints arising as a result of *waitFor*(τ) instructions, an implementation of a cc-Golog interpreter must have a temporal reasoning component. The task of this component is to infer how different constraints combine, for example to infer that $T1 = T2$ follows from $T1 \leq T2 \wedge T2 \leq T1$. Similar to [Rei98], our implementation relies on a logic programming language with a built-in constraint solving capacity, namely on the ECRC Common Logic Programming System Eclipse and the constraint logic programming library clp(q,r) [Hol95]. clp(q,r) allows the specification of linear constraints on PROLOG variables in the style of {`T >= T1, T <= T2`}, and provides the predicate `minimize(T)` to minimize the value of the variable `T` with respect to its constraints. The following Eclipse output illustrates the use of clp(q,r):

```
[eclipse 8]: {2*X+Y >= 16, X+2*Y >= 11, Z = 30*X+50*Y},
             minimize(Z).

  X = 7
  Y = 2
  Z = 310
  yes.
```

Using clp(q,r), we can specify the value of the *t-function*s `constant` and `linear`, which implement the functions *constant* and *linear* from Section 4.1.3, through the following clauses:

```
val(constant(CONST),V,T) :- {V = CONST, T >= 0}.

val(linear(FROM,VEL,T0),V,T) :- {V = FROM+VEL*(T-T0)}.
```

8.1.4 The cc-Golog Interpreter

The following PROLOG clauses, defining the predicates `trans/4`, `final/2`, `trans*/4` and `do/3`, implement the predicates *Trans*, *Final*, *Trans** and *Do*, respectively. They make use of the predicates `holds/2`, `ltp/3`, `earlier/2`, `earliereq/2`, and `restoreFF/5`. `holds(P,S)` is used to evaluate conditions in tests, while-loops and conditionals, `ltp(TForm,S,T)` to determine the least time point of a *t-form*, and `earlier(S1,S2)` respectively `earliereq(S1,S2)` to determine whether the starting time of `S1` is less than `S2`, respectively less or equal `S2`.

Finally, `restoreFF(FormalArgs,Args,Body,S,BodyS)` is used to replace a list of formal arguments `FormalArgs` in a procedure body or primitive action `Body` by the value of the actual arguments `Args` in situation `S`. The actual arguments `Args` may includes functional fluents, which will first be evaluated in `S` before being inserted in place of the formal arguments.[3] We remark that `restoreFF/5` is not only used to evaluate the arguments of a procedure call, but also to evaluate the arguments of primitive actions.

```
% Definition of Final
final(nil,_).
final([],_).
final([A|L],S) :- final(A,S), final(L,S).
final(if(P,A1,A2),S) :-
  holds(P,S), final(A1,S); holds(not(P),S), final(A2,S).
final(while(P,A),S) :- holds(not(P),S); final(A,S).
final(withCtrl(P,A),S) :- holds(P,S), final(A,S).
final(conc(A1,A2),S) :- final(A1,S); final(A2,S).

% Definition of Trans
trans(A,S,nil,do(Ainst,S)) :-
        poss(A,S),
        A=..[Name|Args],
        restoreFF(Args,Args,A,S,Ainst).

trans(waitFor(TF),S,nil,SS,S)) :-
        holds(online,S) -> start(S,T), holdsTForm(TF,S,T), SS = S;
            ltp(P,S,T), SS=do(waitFor(TF),S).

trans(P?,S,nil,S) :-
  holds(P,S).

trans([A|L],S,R,S1) :-
  final(A,S), trans(L,S,R,S1);
  trans(A,S,R1,S1), R=[R1|L].

trans(if(P,A1,A2),S,R,S1) :-
  holds(P,S), trans(A1,S,R,S1);
  holds(not(P),S), trans(A2,S,R,S1).

trans(while(P,A),S,[R,while(P,A)],S1) :-
  holds(P,S), trans(A,S,R,S1).

trans(withCtrl(P,A),S,RR,SS) :-
        holds(P,S), trans(A,S,R,SS), RR = withCtrl(P,R).
```

[3]To illustrate the meaning of `restoreFF/5`, let us consider the example of an elevator controller and assume that we have a procedure `serveFloor` with formal parameter `f` whose purpose is to serve floor `f`. Furthermore, let us assume that we have defined a functional fluent `nextFloor`. Then, if `serveFloor` is called in a situation `s`, `restoreFF([f],[nextFloor],serveFloorBody,s,BodyS)` is invoked by the interpreter to replace all occurrences of `f` in `serveFloorBody` (the body of `serveFloor`) by the value of `nextFloor` in `s`.

```
trans(conc(A1,A2),S,RR,SS) :-
         not final(A1,S), not final(A2,S),
        (trans(A1,S,R1,SS), RR=conc(R1,A2),
           (trans(A2,S,R2,S2) -> earliereq(SS,S2);true);
         trans(A2,S,R2,SS), RR=conc(A1,R2),
           (trans(A1,S,R1,S1) -> earlier(SS,S1);true)).

trans(Proc,S,R,S1) :-
        Proc =..[ProcName|L],
        proc(ProcName,FormalArgs,Body),
        restoreFF(FormalArgs,L,Body,S,BodyS),
        trans(BodyS,S,R,S1)).

trans(A,S,R,SS) :- proc(A,Body), trans(Body,S,R,SS).

% Definition of Trans* and Do using negation as failure
trans*(A,S,A,S).
trans*(A,S,R,S1) :- trans(A,S,R2,S2), trans*(R2,S2,R,S1).

do(A,S,S1) :- trans*(A,S,R,S1), final(R,S1).
```

Let us now turn to the definition of the predicates `holds/2`, `ltp/3`, `earlier/2`, `earliereq/2`, and `restoreFF/5`. The following clauses defining `holds/2` correspond to those presented in [LRL+97]. They make use of the predicate `sub/4` which implements substitution so that $\texttt{sub}(x, y, t, t')$ means that $t' = t_y^x$.

```
holds(and(P1,P2),S) :- holds(P1,S), holds(P2,S).
holds(or(P1,P2),S) :- holds(P1,S); holds(P2,S).
holds(exists(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
holds(not(P),S) :- not holds(P,S).  /* negation as failure */
```

In order to define the predicate `ltp/3`, we first have to specify how we evaluate *t-form*s, i.e. temporal conditions. For that purpose, we define the predicate $\texttt{holdsTForm}(\tau, s, t)$, which implements $\tau[s, t]$. If `TForm` is primitive, i.e. a PROLOG-term `cf >= x` or `cf =< x`, where `cf` is a continuous fluent and `x` a number, `holdsTForm(TForm,S,T)` makes use of `hasValCF/3` and `val/3` to determine the value `V` of the *t-form* `TForm` in situation `S` at time `T`, and `V` is constrained to be at least `X`, respectively at most `X`. On the other hand, if `TForm` is a composite expression $\texttt{and(tf}_1\texttt{,tf}_2\texttt{)}$ or $\texttt{or(tf}_1\texttt{,tf}_2\texttt{)}$, `holdsTForm` is defined recursively.

```
%Test if a tForm holds in situation S at time T
holdsTForm(CF>=X,S,T) :- hasValCF(CF,TFunc,S), val(TFunc,V,T), {V >= X}.
holdsTForm(CF=<X,S,T) :- hasValCF(CF,TFunc,S), val(TFunc,V,T), {V =< X}.

holdsTForm(and(P1,P2),S,T) :- holdsTForm(P1,S,T), holdsTForm(P2,S,T).
holdsTForm(or(P1,P2),S,T) :- holdsTForm(P1,S,T); holdsTForm(P2,S,T).
```

Based on `holdsTForm/3`, `ltp(TF,S,T)` is defined by means of the predicate `minimize/1` provided by `clp(q,r)`. Intuitively, `ltp(TF,S,T)` is true if `T` is the least time point after the start of `S` where `TF` becomes true.

```
ltp(TF,S,T) :-
        holdsTForm(TF,S,T), start(S,T0), {T >= T0}, minimize(T).
```

`earlier/2` and `earliereq/2` are defined in terms of `start/2`, which implements the function *start*, mapping a situation to its starting time. The term `ccUpdate([`x_1`,...,`x_n`],T)` used in the definition of `start/2` refers to exogenous action *ccUpdate*, introduced in Section 5.1.1 to update the value of the continuous fluents during on-line execution.

```
earliereq(S1,S2) :-
        start(S1,T1), start(S2,T2), T1 =< T2.
earlier(S1,S2) :-
        start(S1,T1), start(S2,T2), T1 < T2.

start(s0,0).
start(do(A,S),T) :-
        A=waitFor(TF), ltp(TF,S,T);
        A=ccUpdate(_,T);
        not A=waitFor(_), not A=ccUpdate(_,_), start(S,T).
```

Next, we define `restoreFF(FormalArgs,Args,Body,S,BodyS)`, which takes a program `Body` and replaces each formal argument x_i in the list `FormalArgs` by the value of the corresponding argument a_i in the list `Args`, evaluated in situation `S`. The result is the (instantiated) program `BodyS`. `restoreFF/5` makes use of the predicates `sub_all/4` and `evalFF/3`. The predicate `sub_all/4` is similar to `sub/4`, but substitutes a set of variables instead of a single variable; thus, `sub_all`$(\vec{x}, \vec{y}, t, t')$ means that $t' = t^{\vec{x}}_{\vec{y}}$. The predicate `evalFF(L,S,LS)` takes a list of expressions `L`, evaluates each element in situation `S`, and returns the list of resulting values `LS`.

```
restoreFF(FormalArgs,Args,Body,S,BodyS) :-
        evalFF(Args,S,ArgsS),
        sub_all(FormalArgs,ArgsS,Body,BodyS), !.

evalFF([],S,[]).

evalFF([FF|L],S,[FFInst|LInst]) :-
        hasValFFExp(FF,FFInst,S),
        evalFF(L,S,LInst).
```

Finally, `evalFF/3` makes use of `hasValFFExp/3` to determine the value of an expressions involving functional fluents. In particular, if `FFExp` is a functional fluent or an expression f_1+f_2, f_1-f_2, f_1*f_2, f_1/f_2, where f_1 and f_2 are numbers or numerical functional fluents, then `hasValFFExp(FFExp,S,FFExpS)` determines the value `FFExpS` that `FFExp` has in `S`.

```
% Determine the value of complex expressions
hasValFFExp(FF1+FF2,FFInst,S) :- !,
        hasValFFExp(FF1,FFI1,S),
        hasValFFExp(FF2,FFI2,S),
        FFInst is FFI1 + FFI2.
```

```
% Analogously for -,*,/
% ...

% Determine the value of primitive expressions
hasValFFExp(FF,FFInst,S) :-
        isFF(FF) -> hasValFF(FF,FFInst,S); FFInst=FF.
```

8.1.5 Experimental Results

In order to evaluate the performance of our cc-Golog interpreter, we have applied it to the (slightly modified) example of [BG98], and have compared the resulting runtime with that of [BG98], where the XFRM framework [McD92a, McD94] was used to project RPL plans involving continuous change. In this example, a mobile robot is to deliver letters in the environment depicted in Figure 8.1. At the same time, it has to monitor the state of the environment, in particular it has to check whether doors are open, and eventually has to react if it realizes that a door is open.

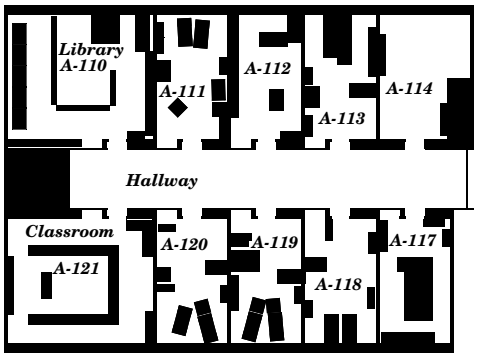


Figure 8.1: Example environment

To approximate the robot's trajectory, we only make use of the *t-function* `toCoords`, which implements the *t-function toCoords* from Section 7.3.3. *toCoords* is a two-dimensional, linear function of time which takes six argument: x_0, y_0, t_0, x_d, y_d, and v. At time t_0, *toCoords* has as value the tuple $\langle x_0, y_0 \rangle$. Starting from there, $toCoords(x_0, y_0, t_0, x_d, y_d, v)$ linearly moves toward $\langle x_d, y_d \rangle$ with velocity v. The following PROLOG clause defines the value of `toCoords` at any time `T`. Note that we take care of the case where $\langle x_0, y_0 \rangle = \langle x_d, y_d \rangle$.

```
val(toCoords(X0,Y0,T0,Xd,Yd,V),[X,Y],T) :-
        X0=Xd, Y0=Yd, X= X0, Y=Y0;
        not (X0=Xd, Y0=Yd),
          Nu2 is (X0-Xd)*(X0-Xd)+(Y0-Yd)*(Y0-Yd),sqrt(Nu2,Nu),
          {X = X0 + (Xd-X0)*(T-T0)*V/Nu,
           Y = Y0 + (Yd-Y0)*(T-T0)*V/Nu}.
```

To represent the robot's position, we make use of the continuous fluent `robotLoc`, whose value is a 2-dimensional *t-function*. In particular, we make use of the *t-function* `toCoords` to approximate any robot trajectory by a piece-wise linear function. `robotLoc` is only affected by the primitive actions `startTo(X,Y)`, `stop` and `ccUpdate([X,Y|_],T)`. Intuitively, `startTo(X,Y)` causes the robot to travel towards position ⟨`X,Y`⟩, `stop` causes it to stay at its current position, and `ccUpdate([X,Y|_],T)`, which represents an update provided by the low-level position estimation process, sets the value of the continuous fluents to the latest position estimate ⟨`X,Y`⟩. For means of simplicity, we assume that the robot always travels at speed 30 m/s. The following clause specifies how `robotLoc` changes its value as a result of the execution of primitive actions.

```
hasValCF(robotLoc,F,do(A,S)):-
```

```
hasValCF(robotLoc,F0,S),
(not A=ccUpdate([X,Y|_],T), not A=startTo(X,Y), not A=stop, F=F0;
 A = ccUpdate([X,Y|_],Tnow), !, V = 30,
     F0 = toCoords(X0,Y0,T0,Xd,Yd,_), F=toCoords(X,Y,Tnow,Xd,Yd,V);
 A = startTo(X,Y), !, start(S,Tnow), V = 30,
     val(F0,[Xnow,Ynow],Tnow), F=toCoords(Xnow,Ynow,Tnow,X,Y,V);
 A = stop, !, start(S,Tnow), V = 1, val(F0,[Xnow,Ynow],Tnow),
     F= toCoords(Xnow,Ynow,Tnow,Xnow,Ynow,V)).
```

As the model of the navigation process, we use the procedure `navProc`, which is a variant of the cc-Golog program *navProc* described in Section 4.3.2. `navProc` becomes active as soon as `reg(destRoom)` is assigned a new value, whereat it calls the procedure `navProcTravelTo` with `reg(destRoom)` as argument.[4] In turn, `navProcTravelTo` approximates the robot's trajectory by polylines, consisting of the starting location, the goal location and a point in front of and behind every passed doorway, similarly to the model of the navigation process used in [BG98]. To do so, it makes use of the procedures `leaveRoom` and `gotoLoc`, and of the functional fluents `entryX/Y(R)`, `exitX/Y(R)`, `centerX/Y(R)`. While the value of these functional fluents correspond to the X respectively Y coordinates of the different intermediate positions, the procedure `leaveRoom` is used to simulated that the robot is leaving a room. Finally, `navProcTravelTo` and `leaveRoom` make use of the procedure `gotoLoc`, which essentially executes a `startTo(x,y)` action and waits until the position $\langle$`x,y`$\rangle$ is reached.

```
proc(navProc,[],
   while(true,
        [and(not(currentRoom=reg(destRoom)),not(reg(destRoom)=nil))?,
         navProcTravelTo(reg(destRoom))])).

proc(navProcTravelTo,[r],
        [conc([if(not(currentRoom=nil),
                    leaveRoom(currentRoom)),
               gotoLoc(entryX(r),entryY(r)),
               gotoLoc(exitX(r),exitY(r)),
               gotoLoc(centerX(r),centerY(r)),
               stop,
               reply(arrivedAt,r)],
              [not(reg(destRoom)=r)?,stop])]).

proc(leaveRoom,[r],
        [gotoLoc(exitX(r),exitY(r)),
         gotoLoc(entryX(r),entryY(r))]).
```

In order to determine the opening state of a door, our example robot can activate a special low-level process (see [SB00] for a possible realization of such a low-level door state estimation process). However, this low-level process can only determine the opening state of a door if the robot is near that door. That is, a high-level plan must only activate the door state

[4] We assume that `reg` has been declared as a functional fluent, using the PROLOG clause `isFF(reg(_))`. Additionally, we assume initial and successor state axiom for `reg` by means of PROLOG clauses defining `hasValFF(reg(Id),Val,s0)` and `hasValFF(reg(Id),Val,S)`.

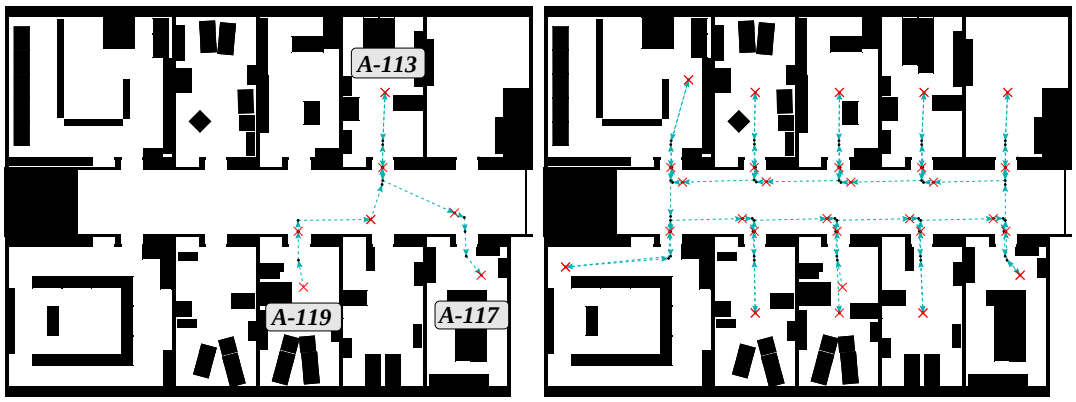
estimation process if it is near a door. To determine whether the robot is near a door at a certain time t in a certain situation s, we make use of the *t-form*-macro `nearDoor(D)` which verifies that the value of `robotLoc` in situation s at time t is a position which is near door `D`.

Some Example Plans Let us now consider an example high-level robot plan, adapted from [BG98]. The robot is to deliver mail to room A-118. At the same time, it has to monitor the state of the environment, that is, it has to check whether doors are open. As soon as it realizes that the door to A-113 is open, it has to interrupt its actual delivery in order to deliver an urgent letter to A-113. This can be specified as a policy that leads the robot inside A-113 as soon as the opportunity is recognized. The following cc-Golog plan makes use of the procedures `checkDoor` and `gotoRoom`. While `checkDoor` is used to activate the door state estimation process, `gotoRoom` implements the procedure *gotoRoom* from Section 4.3.4. The program also includes `say` actions, which are effectless and are merely used to provide an intuition of the "side-effects" of the procedure.

```
proc(aips98Ex,
     withPol(whenever(inHallway,
       [say(enterHW),
         conc(
           [whenever(nearDoor(a-114+117),
             [checkDoor(a114),checkDoor(a117),false?])],
           [whenever(nearDoor(a-113+118),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-112+119),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-111+120),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-110+121),
             [checkDoor(a113),checkDoor(a118),false?])],
           [waitFor(leftHallway),say(leftHW)])]),
        withPol([reg(useOpp)=yes?,gotoRoom(a113),deliverUrgentMail],
          [gotoRoom(a118),giveMail(gerhard)]))).
```

The outer policy is activated whenever the robot enters the hallway, and deactivated when the robot leaves the hallway. It concurrently monitors whether the robot reaches a location near two opposite doors, at which point it checks whether the doors are open or not. The `false?` test executed after each `checkDoor` causes the corresponding thread to block forever, which means that while the robot stays on the floor, it will not check this door again. However, if the robot leaves the hallway and thereafter enters it again, the whole outer policy is activated once again, and thus will re-check the doors it comes close to.

The role of the procedure `checkDoor` is to assign the value `yes` to the register `useOpp` if A-113 is detected to be open. This results in an activation of the inner policy, which is waiting for `reg(useOpp)=yes`. The purpose of this policy is to use the opportunity to enter A-113 as soon as possible. Figure 8.2 (left) illustrates the projected trajectory starting in Room A-119, assuming that the door to A-113 is indeed open. In this projection, we simply modeled `checkDoor` as executing `reply(useOpp,yes)` if and only if it is activated with `a-113` as argument.



(a) Urgent delivery (b) Long trajectory

Figure 8.2: Projected Execution Scenarios

We also used our cc-Golog implementation to project the following plan, which results in a longer trajectory through all rooms.

```
proc(longTraj,
     withPol(whenever(inHallway,
       [say(enterHW),
         conc(
           [whenever(nearDoor(a-114+117),
             [checkDoor(a114),checkDoor(a117),false?])],
           [whenever(nearDoor(a-113+118),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-112+119),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-111+120),
             [checkDoor(a113),checkDoor(a118),false?])],
           [whenever(nearDoor(a-110+121),
             [checkDoor(a113),checkDoor(a118),false?])],
           [waitFor(leftHallway),say(leftHW)])]),

          [gotoRoom(a-114),say("hello a-114"),
           gotoRoom(a-113),say("hello a-113"),
           gotoRoom(a-112),say("hello a-112"),
           gotoRoom(a-111),say("hello a-111"),
           gotoRoom(a-110),say("hello a-110"),
           gotoRoom(a-117),say("hello a-117"),
           gotoRoom(a-118),say("hello a-118"),
           gotoRoom(a-119),say("hello a-119"),
           gotoRoom(a-120),say("hello a-120"),
           gotoRoom(a-121),say("hello a-121")
          ])))))
```

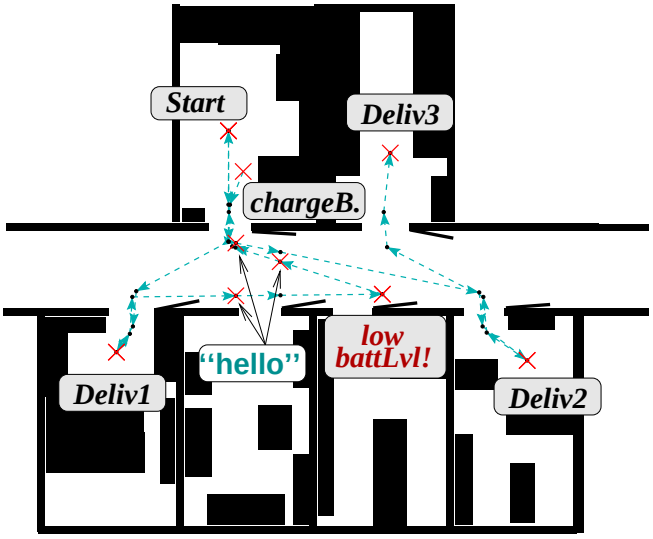


Figure 8.3: Projection of `introEx`

The projection of the above plan is illustrated in Figure 8.2 (right). Finally, we used our cc-Golog implementation to project the following plan `introEx`, which is adapted from the example plan Π_{intro} in Section 4.3.4. We assume that there are three letters to be delivered.

```
proc(introEx,[],
     withPol([waitFor(battLevel=<46),
                 grabWheels,chargeBatteries,releaseWheels],
       withPol([whenever(nearDoor(r6213),
                   [say(hello),waitFor(notNearDoor(r6213))])],
         withCtrl(wheels,
           [gotoRoom(r6212),giveCoffee(ralf),
            gotoRoom(r6214),giveCoffee(guenter),
            gotoRoom(r6205),giveCoffee(sascha)
            ])))).
```

The projection of the example plan `introEx` is illustrated in Figure 8.3, assuming that the delivery is once interrupted by low battery voltage. Figure 8.4 shows the time it took to generate a projection of the example plans using cc-Golog respectively XFRM, together with the number of actions respectively events occurring in the projection. Both cc-Golog and XFRM run on a Linux Pentium III 500Mhz Workstation, under Allegro Common Lisp Trial Edition 5.0 respectively under Eclipse 4.2. Regarding these (quite natural) examples, our cc-Golog

Problem:	cc-Golog	XFRM
`introEx`	0.45 s / 115 acts	-[5]
`aips98Ex`	0.74 s / 82 acts	2.7 s / 105 evs
`longTraj`	3.69 s / 370 acts	15.7 s/ 488 evs

Figure 8.4: Runtime in seconds

implementation appears to be much faster than XFRM. We believe that cc-Golog owes this somewhat surprising advantage to the fact that it lends itself to a simple implementation with little overhead, while XFRM relies on a rather complex implementation involving many thousand lines of Lisp code. However, it turns out that when we consider more complex plans involving a larger number of concurrent sub-plans executing *waitFor* actions, XFRM outperforms our cc-Golog implementation. To study how the performance of cc-Golog respectively of XFRM scales up to more complex plans, we consider versions of `longTraj` where the outer policy (which is waiting for the robot to come close to a door) is *repeated* several times. That is, we considered plans of the form:

`withPol(whenever(inHallway,`*door_policy*,
`withPol(whenever(inHallway,`*door_policy*,
`withPol(whenever(inHallway,`*door_policy*,
...
goto_all_rooms)...)

where *door_policy* stands for the outer policy of `longTraj` and *goto_all_rooms* stands for the sequence of `gotoRoom` procedure calls. In particular, we defined a set of procedures $\texttt{longTraj}^i$, where the index i represents the number of nested *door_policy*'s. Although these plans all result in (approximately) the same behavior, they are well suited to study how the performance of an implementation scales to a larger number of concurrent sub-plans involving *waitFor* actions. As Figure 8.5 illustrates, the XFRM system scales much better than our implementation, which is (probably) due to the fact that XFRM employs special-purpose routines for dealing with blocked threads.

Problem:	cc-Golog	XFRM
`longTraj`	3.69 s	15.7 s
$\texttt{longTraj}^2$	13.23 s	20.7 s
$\texttt{longTraj}^3$	35.37 s	27.3 s
$\texttt{longTraj}^4$	84.14 s	32.8 s

Figure 8.5: Runtime in seconds

We remark that the above examples where XFRM outperforms cc-Golog are rather pathological. Nevertheless, the most important difference between cc-Golog and XFRM is that the cc-Golog implementation is firmly based on a logical specification, while XFRM relies on the procedural semantics of the RPL interpreter.

8.2 Running cc-Golog on a Real Robot

So far, our cc-Golog interpreter is only suitable for the projection of cc-Golog plans. In order to allow the actual on-line execution of cc-Golog plans, we need a run-time system that couples cc-Golog to a real robot. In particular, the run-time system has to handle *send* actions issued by the high-level controller (by eventually activating low-level processes), has to process the

[5]We did not use XFRM to project the introductory example because in its present form XFRM has no model of the batteries voltage level.

messages received from sensor processes (by mapping them to exogenous *reply* actions), and has to realized the link between *waitFor*-actions and continuous low-level processes.

In this section, we describe an implementation of a run-time system that couples the cc-Golog interpreter presented in the last section to the BeeSoft execution system [BBC+95, BCF+00], and we show how this run-time system can be used to control the mobile robot CARL, an RWI B21 robot (Fig. 8.6). CARL is equipped with a laser range finder, 24 sonars and 56 infra-red sensors and four buttons, which can for example be used to confirm a receipt or to choose among different options.

Figure 8.6: The robot CARL

8.2.1 The BeeSoft System

The BeeSoft system is a state-of-the-art basic-task execution system, which has successfully been used to control several robots over extended periods of time. Among others, it has been used to control the museum tourguides RHINO [BBC+95, BCF+00] and MINERVA [TBB+99]. The BeeSoft package consist of several modules which run asynchronously and communicate using the TCX communication library [Fed93]. The overall architecture is illustrated in Figure 8.7. Here, we will only briefly go over the most important modules of the BeeSoft system.

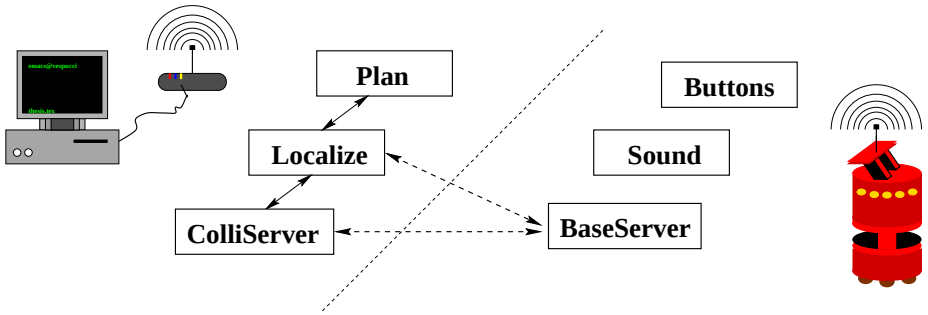


Figure 8.7: Overview of the BeeSoft Architecture

BASESERVER At the bottom of the architecture, a hardware-interface operates the robot's motors and physical sensors.

COLLISERVER The hardware-interface module is used by a collision avoidance system, which also makes use of laser scans and sonar readings to ensures that the robot does not bump into obstacles, following the "dynamic window approach" [FBT97].

LOCALIZE The sensor scans are also used by the Markov localization module [BFHS96, FBDT99], which makes use of a occupancy map of the environment and a model of the sensors to maintain a probability distribution over the possible positions of the robot. Recent experiments have shown that Markov-localization is able to robustly track the position of the robot over an extended period of time [GBFK98, FBT99, BCF+00].

PLAN Navigation is accomplished using a path planer [TBB+98], which is based on an occupancy grid representation of the environment. Given the robot's location and a (sequence of) destination point(s), the path planer makes use of "value iteration" [Bel57] to determine a path from the actual position to the destination. The actual navigation along this path is accomplished using the collision avoidance module.

SOUND and **BUTTONS** Finally, it is possible to play sound files and synthesize speech via a standard sound-card, and to activate and query the robot's buttons.

8.2.2 The Link between cc-Golog and BeeSoft

To couple our PROLOG implementation of cc-Golog to the BeeSoft execution system, we rely on the High-Level Interface HLI [HBL98], which has already been used as link between the BeeSoft system and plain GOLOG in the RHINO museum tour-guide project [BCF+00]. Roughly, HLI provides a uniform PROLOG interface to the different modules of BeeSoft, which allows the parameterized activation of modules along with a monitoring component which provides status updates about the state of execution of the activated modules.

Let us now go over the main execution loop of our cc-Golog execution system ccx. In our PROLOG implementation, the on-line execution of a high-level plan `A` is accomplished by the procedure `ccx/3`. If `ccx(A,S,Exec)` is invoked, where `S` is a situation term representing the actual situation and `A` a term representing the high-level plan to be executed, then `ccx/3` results in the on-line execution of `A`. Upon completion of `ccx/3`, `Exec` is bound to a situation term which represents the resulting on-line execution trace. For example, to start the execution of the plan `introEx`, the high-level controller would call `ccx(introEx,s0,ExecTrace)`.

```
ccx(A,S,Exec) :- final(A,S), !, S=Exec.
ccx(A,S,Exec) :-
        exoAction(E,S), !, ccx(A,do(E,S),Exec);
        trans(A,S,R,SS),
           (SS = do(send(Id,Val),S), execute(send(Id,Val)); true), !,
           ccx(R,SS,Exec));
        not trans(A,S,R,SS), sleepTillNextExog, !, ccx(A,S,Exec).
```

If `A` is final in `S`, the execution of ccx ends. Else, ccx first checks whether an exogenous action `E` has append (either a *ccUpdate* or a *reply*). This is realized by a call to `exoAction/2`, which asks HLI if a new message has been received, and if so maps the message to an exogenous action `E`. If there is such an action `E`, the execution is pursued using `do(E,S)` as actual situation. Else, ccx determines whether there is a transition leading from the actual situation and the cc-Golog plan under execution to a successor configuration $\langle \mathtt{R}, \mathtt{SS} \rangle$. If the transition involves the execution of a *send* action, ccx calls the procedure `execute/1` which invokes HLI to communicate this command to the appropriate BeeSoft module; thereafter, the execution of the remaining plan `R` is pursued in `SS`. Finally, if the plan is blocked, i.e. there is no transition to a successor, then ccx calls `sleepTillNextExog/0` to wait until an exogenous event occurs, whereupon it resumes execution. We remark that ccx's architecture, illustrated in Figure 8.8, corresponds to the overall kind of architecture assumed in Section 4.3.

Using our cc-Golog run-time system ccx, we executed the cc-Golog plan `introEx` from Section 8.1.5 on the mobile robot CARL in the north floor of the Computer Science Department V at Aachen University of Technology. Figure 8.9 illustrates the behavior generated

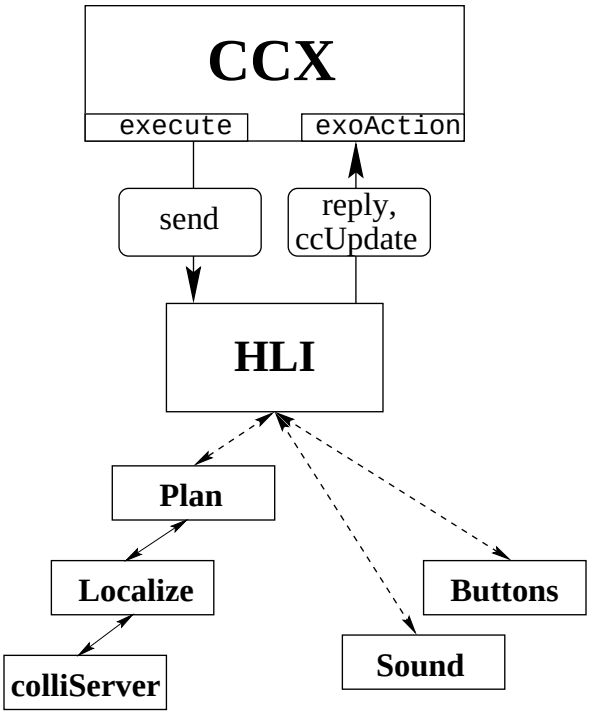


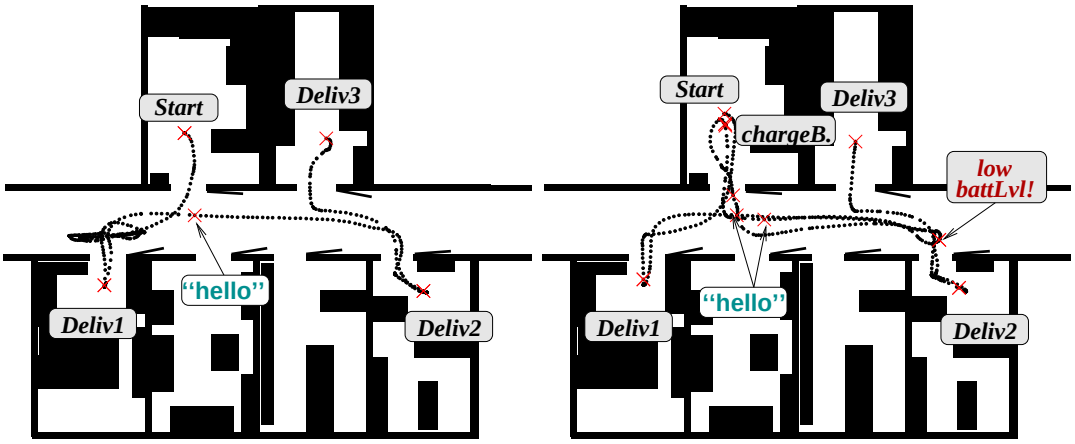
Figure 8.8: The Architecture coupling cc-Golog to BeeSoft

by `introEx` in two different runs. While the left sub-figure shows an on-line execution trace where the delivery is not interrupted by low battery voltage, the right sub-figure shows an execution trace where the battery level drops dangerously low during delivery, and CARL first travels to Room 6204 to recharge its batteries before it completes the delivery.

8.2.3 Interleaving On-Line Execution and Projection

Next, we consider an example where interleaving projection and on-line execution is useful in specifying intelligent robot behavior. Again, our robot is to deliver letters. Additionally, it has to be back in Room 6204 by 15:30, where a practical course will start. In this scenario, a reasonable high-level behavior would be to deliver letters until the time is not sufficient for another delivery (but yet sufficient to travel to Room 6204), at which point the robot would start moving to Room 6204. Using a time-bound projection test telling us whether the robot will be able to deliver another letter *and* to travel to Room 6204 before 15:30, this behavior can (informally) be turned into the following high-level plan: "if projection indicates that there is enough time to deliver another letter, then do so; else, travel to Room 6204."

Before we can turn to the formal specification of this plan, we first have to show how our cc-Golog implementation can be extended to account for time-bounded projection tests (cf. Section 5.2.2). In particular, we introduce the condition `lookahead` which implements the projection test Lookahead discussed in Section 5.2.2. `lookahead(Phi,T,A,LL)` is true if the projection of the plan `A` with time-bound `T`, using `LL` as model of the low-level processes, results in a situation where `Phi` holds. As it turns out, the evaluation of projection tests does not cause any difficulties in PROLOG. In fact, we only need to add a new `holds/2` clause to those already listed in Section 8.1.4.



(a) Sufficient Battery Voltage (b) Low Battery Voltage

Figure 8.9: On-Line Execution Scenarios of `introEx`

```
holds(lookahead(Phi,T,A,LL),S) :-
        timeboundDo(withPol(LL,A),S,T,LL,SS),
        holds(Phi,SS).
```

Note that the low-level processes are taken into account during projection by concurrently simulating `LL` and `A`. To perform a time-bounded projection, the `lookahead` test makes use of the predicate `timeboundDo/4`. `timeboundDo(A,S,T,S1)` is defined similarly to `do(A,S,S1)` (cf. Section 8.1), but additionally verifies that the simulation ends if a situation `S1` is reached whose successor has a starting time beyond `T`.

```
timeboundDo(A,S,T,S) :- final(A,S).

timeboundDo(A,S,T,S1) :-
          trans(A,S,R2,S2), start(S2,T2),
          (T2 > T, S1 = S;
           T2 =< T, timeboundDo(R2,S2,T,S1)).
```

Now that we have implemented the projection test `lookahead`, we are settled for the specification of a plan realizing the behavior informally discussed at the beginning of this sub-section. We begin with a specification of the domain. To represent the letters to be delivered, we make use of the fluent `letter(L,Origin,Destination)`, which is true if letter `L` is to be deliver from `Origin` to `Destination`. The fluent `letter` is only affected by the primitive action `deliver(L)`, as specified by the following clauses:

```
holds(letter(L,Origin,Dest),s0) :-
        [L,Origin,Dest] = [l1,r6213a,r6213];
        [L,Origin,Dest] = [l2,r6214,r6212];
        [L,Origin,Dest] = [l3,r6213,r6205].

holds(letter(L,Origin,Dest),do(A,S)) :-
        holdsF(letter(L,Origin,Dest),S), not A = deliver(L).
```

To deliver a letter `l`, the high-level controller can invoke the procedure `deliverLetter` with `l` as argument. `deliverLetter` makes use of the procedure `gotoRoom(r)` to travel to a room `t`, and of the functional fluents `lOrig(l)` and `lDest(l)`, whose value is the origin respectively the destination of letter `l`.

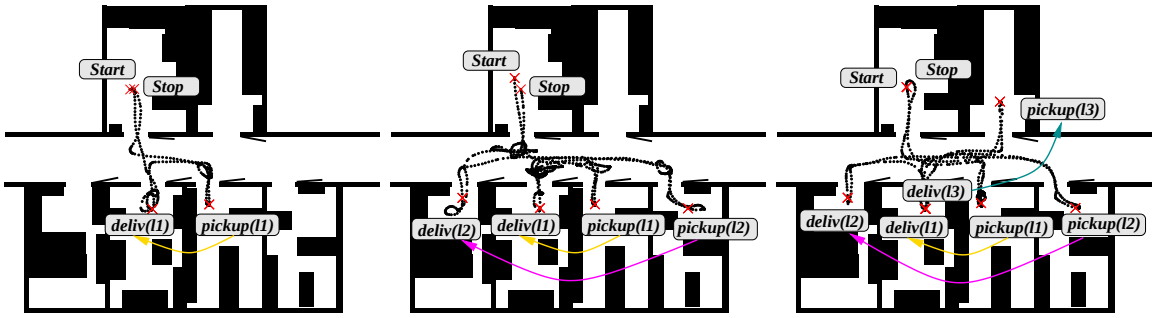
```
proc(delivLetter,[l],
        [gotoRoom(lOrig(l)),
         pickup(l),
         gotoRoom(lDest(l)),
         deliver(l)]).

isFF(nextLetter).
isFF(lOrig(L)).
isFF(lDest(L)).

hasValFF(lOrig(L),V,S) :- (holds(letter(L,V,_),S), ! ; V = nil).
hasValFF(lDest(L),V,S) :- (holds(letter(L,_,V),S), ! ; V = nil).
```

Now that we have introduced the projection test `lookahead` and have specified the example domain, we can turn the informal specification "if projection indicates that there is enough time to deliver another letter, then do so; else, travel to Room 6204" into the cc-Golog plan `deliverUntil`. `deliverUntil` takes as argument the time `t` at which the robot has to be back in Room 6204. If there are no letters to be delivered, `deliverUntil` directly causes the robot to travel to Room 6204. If there still are letters to be delivered, `deliverUntil` determines – by means of a projection test – if it is possible to deliver the next letter *and* to be back in Room 6204 by time `t`. If so, it delivers the next letter and re-evaluates the new situation. Finally, if there is not enough time to deliver another letter, `deliverUntil` causes the robot to travel directly to Room 6204:

```
proc(delivUntil,[t],
        if(not(exists(l,exists(o,exists(d,letter(l,o,d))))),
            gotoRoom(r6204), % Delivery Completed
          if(lookahead(robotInRoom(r6204),t,
                       [deliverLetter(nextLetter),gotoRoom(r6204)],
                       navProc),
            % If enough time, then continue delivery
              [deliverLetter(nextLetter),delivUntil(t)],
            % Else stop delivery
               gotoRoom(r6204)))).
```

Here, `robotInRoom(R)` is a macro, which is true if the robot's position lies within room `R`, and `nextLetter` is a functional fluent (defined in terms of the fluent `letter`) whose value is the next letter to be delivered.

Figure 8.10 illustrates three different runs of `deliverUntil`. In the execution trace shown in the left sub-figure, CARL is only left 80 seconds to perform delivery, which only suffices to deliver one letter from Room 6213a to Room 6213. In the execution trace shown in the middle sub-figure, the robot can perform for 350 seconds before it has to be back in Room 6204, which allows it to deliver another letter from Room 6214 to Room 6212. Finally, the right sub-figure



(a) 80 seconds remaining (b) 350 seconds remaining (c) 950 seconds remaining

Figure 8.10: Time-Bound Delivery

shows an execution trace where the robot has enough time to deliver all letters. We remark that all limited projection tests involved in these examples were computed in less than 0.15 seconds, running on a Pentium III 500Mhz Linux Workstation.

Finally, we remark that in examples like the above, the reliability of the high-level controller heavily depends on the accuracy of the model of the navigation process used in the projection test. However, our execution scheme interleaving on-line execution and projection is still much more robust than the unreflected execution of a plan determined completely off-line, because each projection test takes into account the actual history encountered so far, thus adapting the robot's behavior to previous delays.

8.3 A pGOLOG Interpreter in PROLOG

Let us now turn to the implementation of a pGOLOG interpreter in PROLOG. We use the same PROLOG terms as legal representations of pGOLOG programs as for cc-Golog (cf. Section 8.1.2), together with the following construct which represents the probabilistic branching instruction prob:

- `prob(p,`$\mathtt{a}_1$`,`$\mathtt{a}_2$`)`, where `p` is a number between 0 and 1, and $\mathtt{a}_1$ and $\mathtt{a}_2$ are programs.

In addition to the clauses listed in Section 8.1.1, which are used to define a non-probabilistic domain, our pGOLOG implementation requires the following parts to specify the robot's probabilistic epistemic state:

- A collection of clauses defining the predicate `initialSit(`$\mathtt{s}_i$`,`$\mathtt{p}_i$`)`, which is used to specify all situations $\mathtt{s}_i$ which the robot considers possible initially, together with their initial probability $\mathtt{p}_i$.
- A collection of clauses defining the predicate `initialLL(`$\mathtt{s}_i$`,`$\mathtt{ll}_i$`)`. Here, $\mathtt{ll}_i$ is a pGOLOG program modeling the low-level processes in situation $\mathtt{s}_i$. The idea is that `initialSit/2` together with `initialLL/2` specify the initial value of the fluent *pll* (cf. Section 7.1.2).
- A collection of clauses defining the predicate `holdsF(F,`$\mathtt{s}_i$`)`, defining which fluents `F` are true in the possible situations $\mathtt{s}_i$. Similarly, we require clauses `hasValCF(CF,TFunc,`$\mathtt{s}_i$`)` and `hasValFF(FF,V,`$\mathtt{s}_i$`)` to define the value of the continuous and functional fluents.

8.3.1 Probabilistic Projection

The following PROLOG clauses defining the predicates `transPr/5`, `transPr*/5` and `doPr/4` implement, respectively, the functions *transPr*, *transPr**, and *doPr*. Here, we use the last argument of the predicates to represent the value of the function implemented, i.e. the transition probability. The following clauses make use of the predicates `holdsTForm/3`, `ltp/3`, `start/2`, `holds/2` and `final/2`, which were already discussed in the previous section.

```
% Definition of transPr
transPr(A,S,nil,do(Ainst,S),1) :-
        poss(A,S),
        A=..[Name|Args],
        restoreFF(Args,Args,A,S,Ainst).

transPr(waitFor(TF),S,nil,SS,1) :-
        holds(online,S) -> start(S,T), holdsTForm(TF,S,T), SS = S;
            ltp(P,S,T), SS=do(waitFor(TF),S).

transPr(P?,S,nil,S,1) :-
  holds(P,S).

transPr([A|L],S,R,S1,Prob) :-
        final(A,S) -> transPr(L,S,R,S1,Prob);
            transPr(A,S,R1,S1,Prob), R=[R1|L].

transPr(if(P,A1,A2),S,R,S1,Prob) :-
        holds(P,S) -> transPr(A1,S,R,S1,Prob);
            transPr(A2,S,R,S1,Prob).

transPr(while(P,A),S,[R,while(P,A)],S1,Prob) :-
        holds(P,S), transPr(A,S,R,S1,Prob).

transPr(withCtrl(P,A),S,RR,SS,P) :-
        holds(P,S), transPr(A,S,R,SS,P), RR = withCtrl(P,R).

transPr(conc(A1,A2),S,R,SS,Prob) :-
        not final(A1,S), not final(A2,S), (
            transPr(A1,S,R1,SS,Prob),
            (transPr(A2,S,R2,S2,_) -> earliereq(SS,S2);true),
          R = conc(R1,A2);
            transPr(A2,S,R2,SS,Prob),
            (transPr(A1,S,R1,S1,_) -> earlier(SS,S1);true),
            R = conc(A1,R2)).

transPr(prob(P,A1,A2),S,R,S1,Prob) :-
        Prob is P, R=A1, S1=do(tossHead,S);
        Prob is 1-P, R=A2, S1=do(tossTail,S).
```

```
transPr(Proc,S,R,S1,Prob) :-
        proc(Proc,Body), !, transPr(Body,S,R,S1,Prob).

% Definition of transPr* and doPr using negation as failure
transPr*(A,S,A,S,1).
transPr*(A,S,R,S1,Prob) :-
        transPr(A,S,R2,S2,P1), transPr*(R2,S2,R,S1,P2), Prob is P1*P2.

doPr(A,S,S1,Prob) :- transPr*(A,S,R,S1,Prob), final(R,S1).
```

Based on `doPr/4`, we turn to the definition of `pbel/4`, which implements the term *PBel* from Section 6.3.1. Given a sentence `Phi`, a program `A` and a situation `S`, `pbel(Phi,S,A,P)` calculates the agent's belief `P` that `Phi` will hold after the execution of `A` in situation `S`. While so far the PROLOG axioms followed quite naturally from the logical axioms, at this point our implementation differs from the formal specification given Section 6.3.1. In particular, while the formal specification of *Proj* resorts to second-order logic to specify a summation, our interpreter makes use of PROLOG's all-solution predicate `setof`.

Before we define `pbel/4`, however, we first introduce the predicate `pll/4` which implements the fluent *pll* from Section 7.1.2. `pll(S1,LL1,S,P)` is true if in situation `S`, the robot considers situation `S1` possible with probability `P`, and believes that if the world is in situation `S1` then the low-level processes can be characterized by the pGOLOG program `LL1`. The following clause makes use of the user-defined predicates `initialSit/2` and `initialLL/2` to determine the initial value of `pll/4`.[6]

```
pll(S1,LL1,S,P) :- initial(S), initialSit(S1,P), initialLL(S1,LL1).
```

In the definition of `pbel/4`, we make use of the predicates `getAllTraces/3`, `calcPhiProb/3` and `normFactor/2`. The meaning of these predicates is as follows: Given a program `A` and a list `L` of tuples `[S1,LL1,P1]`, each representing a possible situation together with its associated pGOLOG program and weight, `getAllTraces(L,A,Traces)` determines the list `Traces` of all execution traces that can result from the execution of `A` in an initial configuration in `L`. The elements of `Traces` are tuples `[S,P]`, where `S` is a possible execution trace and `P` its weighted occurrence probability (cf. Section 6.3.1). On the other hand, `calcPhiProb(Phi,Traces,Punorm)` is used to determine the unnormalized weight `Punorm` of all execution scenarios in `Traces` that satisfy condition `Phi`. Finally, `normFactor(L,Norm)` is used to determine the sum of the weights of the elements of list `L`.

```
pbel(Phi,S,A,P) :-
        setof([S1,LL1,P],pll(S1,LL1,S,P),L),
        getAllTraces(L,A,Traces),
        calcPhiProb(Phi,Traces,Punorm),
        normFactor(L,Norm),
        (Norm = 0 -> P = undefined;
            P is Punorm/Norm).
```

After collecting all situations considered possible in the list `L`, `pbel/4` invokes `getAllTraces/3` to get all execution scenarios that can result from the execution of `A` in a situation in `L`. Thereafter, it makes use of `calPhiProb/3` and `normFactor/2` to calculate the normalized belief `P`

[6] In our implementation, the predicate `initial(S)` is true if and only if `S` is `s0` or a situation s_i considered possible initially. Note that as a result, the PROLOG clause specifies that p is Euclidean (cf. Section 6.3.1).

that `Phi` will hold after the execution of `A` in situation `S`. The predicates `getAllTraces/3`, `calPhiProb/3` and `normFactor/2` are implemented as follows:

```
getAllTraces([],A,[]).
getAllTraces([[S1,LL1,P1]|T],A,Traces) :-
        setof([S,Pw],weightedDoPr(withPol(LL1,A),S1,P1,S,Pw),Traces1),
        getAllTraces(T,A,TracesT),
        append(Traces1,TracesT,Traces).

% Weight probability of exec. trace with initial prob. of S1
weightedDoPr(A,S1,P1,SS,Pweighted) :-
        doPr(A,S1,SS,Ptrace), Pweighted is Ptrace*P1.

calcPhiProb(Phi,[],0).
calcPhiProb(Phi,[[S1,P1]|T],P) :-
        holds(Phi,S1), calcPhiProb(Phi,T,P2), P is P1+P2;
        calcPhiProb(Phi,T,P2), not holds(Phi,S1), P is P2.

normFactor([],0).
normFactor([[_,_,P1]|L],N) :- normFactor(L,N1), N is N1+P1.
```

Based on `pbel/4`, it is straightforward to calculate the expected utility of a plan (cf. Sections 6.3.3 and 7.3.2). For that purpose, we define the predicate `eu(S,A,Res)`, which implements the function $EU(\sigma, s)$ (as usual, the last argument of `eu/3` represents the value of the function EU). Note that to make use of `eu(S,A,Res)` the user has to define the fluent `util(U)`, whose value in a situation `S` is the total accumulated utility in `S`.

```
eu(S,A,Res) :-
        findall([V,P],pbel(util(V),S,A,P),L),
        calcEU(L,Res).

calcEU([],0).
calcEU([[V,P]|T],Res) :-
        calcEU(T,Res2), Res is (V*P) + Res2.
```

The expected utility of plan `A` in situation `S` is simply the weighted average of the projected value of the fluent `util` after execution of `A` in `S`.

8.3.2 Belief and Belief Update

Let us now turn to the predicates dealing with belief and belief update. The following PROLOG clauses define the predicate `transPrTo/5`, which implements the function $transPr^{\triangleleft}$. Given a program `L`, a situation `S` and a time limit `T`, `transPrTo(L,S,LL,SS,P)` returns a farthest configuration $\langle$`LL`, `SS`$\rangle$ which can be reached through simulation of `L` in `S` without executing a *reply* action. Additionally, it computes the probability `P` to reach $\langle$`LL`, `SS`$\rangle$ (there may be many farthest configurations).

```
transPrTo(L,S,LL,SS,P) :-
        findall([L1,S1,P1],
```

```
            (transPr(L,S,L1,S1,P1), not S1 = do(reply(_,_),S)
            ),
            Lsucc),
          (Lsucc=[] -> LL=L,SS=S,P=1;
          member([L1,S1,P1],Lsucc),
          transPrTo(L1,S1,LL,SS,P2), P is P1*P2).
```

Unlike $transPr^{\triangleleft}$, `transPrTo/5` is not based on $transPr^{*}$ but instead computes the transitive closure of `transPr` by itself. In particular, `transPrTo/5` computes the set `Lsucc` of all successor configurations of ⟨`L`, `S`⟩ which can be reached without execution of a *reply* action. If this set is empty, then ⟨`L`, `S`⟩ is a farthest configuration. Else, `transPrTo/5` is recursively applied to each element of `Lsucc`.

Based on `transPrTo/5`, we define the successor state axiom for `pll/4`. The following definition follows directly from the successor state axiom for *pll*4 from Section 7.1.3.

```
pll(S1,LL1,do(A,S),P) :-
        pll(S11,LL11,S,P1),
        transPrTo(LL11,S11,L1,SS1,P11),S1=do(A,SS1),
        (A \= reply(_,_), LL1=L1, P is P1 * P11;
        A = reply(_,_), transPr(L1,SS1,LL1,S1,1), P is P1 * P11).
```

Next, we define the predicate `bel/3`, which implements the function *Bel*. Given a situation `S` and a formula `Phi`, `bel(Phi,S,Prob)` calculates the robot's degree of belief that `Phi` holds in `S`. To do so, `bel/3` first determines the set `L` of all situations considered possible in `S`, together with their weight. Thereafter, it calculates `Punorm`, the sum of the weights of all situations in `L` which satisfy `Phi`, using the predicate `calcPhiProb/3` introduced in Section 8.3.1. Finally, it calculates the total sum of the weights of all situations in `L`, in order to determine the normalized belief in `Phi`.

```
bel(Phi,S,Prob) :-
        setof([S1,P],pll(S1,_,S,P),L),
        calcPhiProb(Phi,L,Punorm),
        calcPhiProb(true,L,Norm),
        (Norm = 0 -> Prob = undefined;
            Prob is Punorm/Norm).
```

8.3.3 Epistemic Conditions and bGOLOG Programs

As discussed in Section 6.2.4, high-level bGOLOG plans are restricted in the actions they may execute and in the fluents they may appeal to. In particular, they may only appeal to the robot's beliefs or to a projection test in conditionals, tests, while-loops and withCtrl instructions, and they may only appeal to the epistemic functional fluent *Kwhich* (cf. Section 7.3.1) in primitive actions or procedure calls. We will now describe how epistemic conditions are dealt with in our implementation.

In particular, we have implemented the condition `bel(Phi,Prob)`, which is true if the robot's actual belief in `Phi` is `Prob`, along with the projection test `pbel(Phi,A,Prob)`, which is true if the robot's projected belief that `Phi` will hold after execution of `A` is `Prob`, and the utility test `eu(A,Util)`, which is true if the expected utility of the plan `A` in the actual situation is `Util`. Similar to the projection test implemented in Section 8.2.3, the evaluation

of conditions appealing to the robot's beliefs does not cause any difficulties in PROLOG. We merely have to add another few `holds/2` clauses.

```
holds(bel(reg(Id)=Val,Prob),S) :-
        !, hasValFF(reg(Id),Val,S) -> Prob=1;Prob=0.

holds(bel(Phi,Prob),S) :-
        bel(Phi,S,Prob)),

holds(pbel(Phi,A,Prob),S) :-
        pbel(Phi,S,A,Prob).

holds(eu(A,Util),S) :-
        eu(S,A,Util).
```

The first clause makes use of the fact that *reg* is directly observable to provide an efficient evaluation of belief tests appealing to `reg`. The other clauses simply invoke the corresponding PROLOG predicates to determine the truth value of `bel`, `pbel` and `eu` conditions, respectively.

Similarly, the epistemic functional fluent `kwhich` from Section 7.3.1 can be implemented by adding the following clause to the definition of the predicate `hasValFFExp/3`, which is used to determine the value of expressions involving functional fluents (cf. Section 8.1.4):

```
hasValFFExp(kwhich(FF),FFInst,S) :-
        bel(FF=X,S,1) -> FFInst=X; FFInst=nil.
```

Based on the implementation of epistemic conditions, we can now turn to the representation of bGOLOG plans in PROLOG. We use similar terms as for pGOLOG programs, but additionally require that bGOLOG plans:

- do not include any `prob` or `waitFor` instruction;
- do only appeal to epistemic conditions in tests, conditionals, while-loops and withCtrl constructs;
- do not appeal to functional fluents in primitive actions and procedure calls, except for the epistemic functional fluent `kwhich`;
- and do not execute actions which affect a fluent used to describe the state of the world.

8.3.4 Experimental Results

To evaluate the performance of our pGOLOG implementation, we used it to project the plans $\Pi_{robby1}, \Pi_{robby2}, \Pi_{robby3}$ and Π_{robby4} from Section 6.3.2 in the ship/reject domain. The running time for the projection of the different plans is summarized in the following table, where the interpreter run on a Pentium III 500Mhz Linux Workstation.

Projection - plans appealing only to directly observables				
Domain	ship/reject			
Plan	Π_{robby1}	Π_{robby2}	Π_{robby3}	Π_{robby4}
Runtime in seconds	0.12	0.05	0.09	0.25

Besides the above bGOLOG plans which only appeal to the robot's beliefs regarding the value of the directly observable *reg*, we also projected the truly belief-based plan $\Pi_{forbidden}$ from Section 6.2.4 (a variant of Π_{robby4} which completely avoids the use of *reg* in tests), and the belief-based plans $\Pi_{loopInsp}$ and $\Pi_{loopInsp\&Paint}$ from Section 7.3.1. We also computed the expected utility of the plans Π_{robby4} and $\Pi_{forbidden}$ (cf. Section 6.3.3). As the following table shows, the use of belief tests referring to non-observable fluents significantly reduces the performance of our PROLOG interpreter.

Projection (ii) - belief-based plans					
Mode	Projection			Expected Util.	
Plan	$\Pi_{forbidden}$	$\Pi_{loopInsp}$	$\Pi_{loopInsp\&Paint}$	Π_{robby4}	$\Pi_{forbidden}$
Runtime in sec.	16.79	2.07	23.02	0.3	16.76

This strong increase in runtime compared to the projection of plans which only appeal to the robot's beliefs in *reg* is due to the fact that calculating the robot's beliefs is a much more complex task than determining the value of an ordinary fluent (which suffices to determine the beliefs in *reg* because *reg* is directly observable). Note that to calculate the robot's beliefs the interpreter has to determine all situations considered possible by invoking `pll/4`, which will recurse all the way back to the initial situation. This results in a runtime which is at least linear in the length of the history of the actual situation, and can even become exponential depending on the number of situations considered possible.[7] Furthermore, we remark that even the projection of $\Pi_{loopInsp}$ implies the evaluation of 40 belief tests; the projection of $\Pi_{loopInsp\&Paint}$ even implies the evaluation of 140 belief tests.

Next, we used our implementation to calculate the probability distribution over the robot's updated beliefs in its position in different situations of the BHL domain. In particular, we considered the situations S_{r1} to S_{r6} from Section 7.2. We also computed the robot's beliefs in the widget being flawed in situation S_{ok} and in the widget being undercoated in situation S_{UC} (cf. Sections 7.1.4 and 7.1.1, respectively). The running time for the different situations is summarized in the following table.

Belief								
Domain	BHL						ship/reject	
Situation	S_{r1}	S_{r2}	S_{r3}	S_{r4}	S_{r5}	S_{r6}	S_{ok}	S_{UC}
Runtime in sec.	0.02	0.04	0.07	0.11	0.28	0.73	0.02	0.02

Finally, a promising property of the pGOLOG framework is that it is easily amenable to Monte-Carlo methods (e.g. [Fis96]) for the estimation of the success probability of a pGOLOG program (unless, of course, exact assessment is required). In a nutshell, Monte-Carlo simulation can be achieved by pursuing only one of the branches of a prob instruction depending on the outcome of a random number toss. The appealing property of Monte-Carlo methods is that the number of samples to be considered depends only on the desired precision of the estimate, not on the length of the program. In the domains considered so far, however, we did not feel the need for Monte-Carlo methods.

[7] A promising approach to improve the performance of our implementation would be to use some kind of database progression [LR97] during on-line execution.

8.4 Running pGOLOG on a Real Robot

Similarly to cc-Golog, we have used pGOLOG to control a real robot. In this section, we describe a letter delivery application where a mobile robot is to deal with sensing and probabilistic effects, and show how the pGOLOG framework can be used to specify a simple greedy robot controller which tries to minimize the robot's total expected travel time. Finally, we describe some scenarios where the pGOLOG controller was run on the mobile robot CARL.

8.4.1 An Example Application: Colored Letter Delivery

To illustrate the use of pGOLOG to control a mobile robot, let us consider the following letter delivery application, adapted from [BBG99]: a mobile robot is to deliver letters, which are inside colored envelopes. The envelopes can be either red, yellow, green or blue. The robot must care about the color of the envelopes loaded because whenever it delivers a letter, it has to tell the recipient the color of the envelope of the letter designated for her. The recipient will then pick up the letter and acknowledge the receipt by pressing one of the robot's buttons. In order to avoid ambiguities, the robot should take care not to carry different envelopes with the same color, because this forces the recipient to open different envelopes in order to find out which letter is intended for her. Although we do not treat this as a hard constraint, it causes an undesirable delay. As environment we use the north floor of the Computer Science Department V at Aachen University of Technology, familiar from Section 8.2.

Initially, the robot has only incomplete information about the letters to be delivered. Although each letter delivery request includes the origin and the destination of the letter to be delivered, it does not necessarily provide information about the color of the envelope the letter will be in. In particular, we assume that the letter requests are represented by means of the following registers

- `reg(lReq(L))`. This register is used to specify which letters `L` are to be delivered. If the value of `reg(lReq(L))` is `red, yellow, green` or `blue` this indicates that letter `L` is to be delivered in an envelope of the corresponding color, while the value `unspecified` indicates a request where the envelope's color has not yet been specified.[8]
- `reg(lOrig(L))`. This register is used to specify the room where letter `L` is to be picked up.
- `reg(lDest(L))`. This register is used to specify the destination of letter `L`.

In case of an under-specified letter request, the robot will first be told the color of the letter's envelope when it picks up the letter. For that purpose, the sender will make use of the four buttons of the robot, that is she will press the button with the envelope's color after giving the robot the letter.

The Low-Level Processes More specifically, the low-level execution system provides a low-level process *pickup* which can be used to pick up a letter. The low-level process *pickup* can be activated by executing `send(fork,pickupLetter([L,ColorSpec]))`, where `L` refers

[8] We remark that in principle it would be possible to represent a partially specified request by means of a distribution over possible envelope colors. However, for practical purposes our representation is more appropriate as it results in a computationally simpler reasoning task; see the discussion at the end of this subsection.

to the letter to be picked up and `ColSpec` is either the color of `L`'s envelope or the term `unspecified`. Once activate, *pickup* asks the user to give the robot the letter and thereafter to press a button; if the color of the letter to be picked up is `unspecified`, *pickup* asks the sender to press the button with the envelope's color. Once the user has given the letter to the robot and pressed a button, *pickup* executes a `reply(lLoaded(L),C)` action, where `C` is the color of the envelope loaded.

Similarly, the robot can make use of the low-level process *deliver* to deliver a letter. *deliver* is activated by executing `send(fork,deliverLetter([L,Color]))`, where `L` refers to the letter to be delivered and `Color` to the color of the envelope the letter is in. Once activated, *deliver* asks the recipient to take her letter, and tells her the color of the envelope in which her letter is in. Next, *deliver* waits until the user presses a button to acknowledge the receipt. Finally, it subsequently executes a `reply(lLoaded(L),nil)` and a `reply(lReq(L),nil)` action, accounting for the fact that letter `L` has now been delivered. We remark that the register `reg(lLoaded(L))` can thus be used to determine which letters the robot has currently loaded.

8.4.2 The Link between pGOLOG and BeeSoft

In order to execute bGOLOG plans on a real robot, we have to couple our PROLOG interpreter to the low-level execution system of the robot. As it turns out, adapting the run-time system for cc-Golog described in Section 8.2 to pGOLOG only requires minor changes. The coupling of our pGOLOG interpreter to the BeeSoft execution system is realized by the PROLOG procedure `px/3`. If called with `Prg` bound to a bGOLOG plan and `S` bound to a situation, `px(Prg,S,Exec)` results in the on-line execution of `Prg`, and in `Exec` being bound to the resulting on-line execution trace. `px/3` is implemented in complete analogy to the procedure `ccx/3` described in Section 8.2.2. The only difference is that it makes use of `transPr/5` instead of `trans/4` to determine the transitions caused by the plan.

```
px(Prg,S,Exec) :- final(Prg,S), !, S=Exec.
px(Prg,S,Exec) :-
        exoAction(E,S), !, px(Prg,do(E,S),Exec);
        transPr(Prg,S,R,SS,1),
           (SS = do(send(Id,Val),S), execute(send(Id,Val)); true), !,
           px(R,SS,Exec));
        not transPr(Prg,S,R,SS,1), sleepTillNextExog, !, px(Prg,S,Exec).
```

8.4.3 A Simple Greedy pGOLOG Controller

To illustrate how the pGOLOG framework can be used to design robot controllers that account for probabilistic uncertainty, we will now present a simple greedy pGOLOG controller. Our controller aims at minimizing the total amount of time needed to deliver all letters. In particular, we assume that if the robot delivers a letter whose envelope's color is ambiguous, meaning that it has loaded another envelope with the same color, the recipient will need 30 seconds on average to find the right letter. Note that this means that we simply identify utility with total delivery time, and do not further penalize the robot for unnecessarily disturbing people while asking them to find out the right envelope. Furthermore, we assume that if the robot picks up a letter whose color has not been specified beforehand, all four possible envelope colors have the same probability.

The basic idea is as follows: when the robot has loaded some letters, it can either (a) first pick up another letter before delivering the letters loaded; or (b) first deliver all letters loaded before picking up further letters. In general, the robot could also deliberate over which letter to pick up next. Here, however, we use a quite simple heuristic to determine the letter that is potentially to be picked up next: we sort the pickup places clockwise, and identify the next letter to be picked up with the pickup place next to the robots actual position. For example, if the robot is in Room 6204, we subsequently determine if there is a pickup request for Room 6204, 6205, 6214, 6213a, 6213, and finally 6212, and identify the first request found with the next letter to be picked up. Similarly, if the robot starts delivering letters, it proceeds in a clockwise manner.

The decision which option to take depends on which is faster: (a) first picking up the next letter, then delivering all letters; or (b) first delivering all letters loaded, then picking up and delivering the next letter. To correctly estimate the execution time of the different strategies, the robot has to take into account that an ambiguous delivery will cause a delay of 30 seconds on average. Whether or not the robot should put up with this delay not only depends on the length of the detour that (b) would cause, but also of the probability that (a) will result in a delay: note that the robot has only incomplete information about the colors of the envelopes to be delivered. Although this heuristic might not always lead to an ideal robot behavior, our experiments suggest that in practice it results in quite good schedules.

The Model of the Low-Level Processes Before we can turn this informally described heuristic strategy into a bGOLOG plan, we need a pGOLOG model of all the low-level processes involved. To start with, we use the same model of the navigation process as in Section 8.1.5. Note that this model makes use of the *t-function* `toCoords`, which (as elaborated in Section 7.3.3) enables the projection of the remaining plan at any time during on-line execution, based on the updated belief state of the robot. This feature of the model of the navigation process is important here, because our high-level controller will make use of the epistemic test `eu` (cf. Section 8.3.3) and will thus interleave on-line execution and probabilistic projection.

The effects of the activation of *pickup* is modeled by the procedure `prgPickupLetter`, which takes as arguments the letter `l` to be picked up and the specification of the color of the letter's envelope, `colSpec`. The value of `colSpec` may either be `red, yellow, green, blue` or `unspecified`. The outcome of `prgPickupLetter` depends on whether or not the color of the envelope to be picked up is specified. In particular, if the color is yet unspecified `prgPickupLetter` can result in four different outcomes, which reflects the fact that the user can push each of the four buttons of the robot. Each possible outcome has an occurrence probability of 25%. If the color was specified, `prgPickupLetter` has only one possible outcome. Note that each execution trace of `prgPickupLetter` results in the execution of exactly one `reply(lLoaded(l),C)` action.

```
proc(prgPickupLetter,[l,colSpec],
        if(colSpec=unspecified,
          %Color unspecified; four possible outcomes
          [say("Please give me the letter,
                then press the button with the envelope's color."),
           prob(0.5,prob(0.5,[reply(lLoaded(l),red)],
                             [reply(lLoaded(l),yellow)]),
                    prob(0.5,[reply(lLoaded(l),green)],
```

```
                              [reply(lLoaded(l),blue]))],
            %Color specified
            [say("Please give me the letter, then press any button."),
             if(colSpec=red,[reply(lLoaded(l),red)],
                if(colSpec=yellow,[reply(lLoaded(l),yellow)],
                   if(colSpec=green,[reply(lLoaded(l),green)],
                      if(colSpec=blue,[reply(lLoaded(l),blue)])))))]).
```

To model the low-level process *deliver*, we make use of the procedure `prgDeliverLetter` which takes as arguments the letter `l` to be delivered and the `color` of the letter's envelope. `prgDeliverLetter` accounts for the fact that if the robot has loaded more than one envelope with color `color` the recipient has to find out the right envelope by eventually executing `waitTime(60)` (as elaborated in Section 6.2.3, the purpose of $waitTime(n)$ is to wait for n seconds). Finally, `prgDeliverLetter` results in the execution of `reply(lLoaded(l),nil)` and `reply(lReq(l),nil)`, which tells the high-level controller that Letter `l1` has now been delivered.

```
proc(prgDeliverLetter,[l,color],
        [if(exists(l2,and(reg(lLoaded(l2),color),not(l=l2))),
            %Different envelopes with same color
            [say("Your letter is in one of the",color,"envelopes."),
             prob(0.5,waitTime(0),waitTime(60))],
           %Unambiguous delivery
           say("Your letter is in the",color,"envelope.")),
         reply(lLoaded(l),nil),
         reply(lReq(l),nil)]).
```

Based on these models of the individual low-level processes, the whole low-level execution system is then modeled by a single pGOLOG program, analogously to the treatment in Section 6.2.3.

The High-Level Plans Now that we have a model of the low-level processes involved, we can turn to the specification of the high-level bGOLOG plans. We begin by specifying bGOLOG plans which implement the two possible behaviors the robot will deliberate over, i.e. (a) first pick up the next letter, then deliver all letters loaded; and (b) first deliver all letters loaded, then pick up and deliver the next letter. The two possibilities are implemented by the procedures `firstGet` and `firstDeliv`. Besides appealing to the (directly observable) registers, these procedures make use of the `kwhich` construct to evaluate the functional fluents `nextPickup` and `nextDeliv`. `nextPickup` and `nextDeliv` are defined functional fluent whose value is the name of the next room where a letter is to be picked up respectively to be delivered. As mentioned earlier, the value of these functional fluents is defined based on a clockwise ordering of the rooms.

```
proc(firstGet,[],
     [if(not(kwhich(nextPickup)=nil),
         getLetter(kwhich(nextPickup))),
      delivAllLoaded]).
```

```
proc(firstDeliv,[],
     [delivAllLoaded,
      if(not(kwhich(nextPickup)=nil),
         getLetter(kwhich(nextPickup))),
      delivAllLoaded]).
```

The above procedures make use of the procedures `getLetter` and `delivAllLoaded` to get the next letter respectively to deliver all letters loaded. `getLetter` first calls `gotoRoom` to move to the right place and then activates the low-level process *pickup* by means of a `send(fork,pickupLetter(l,cs))` action. `delivAllLoaded` invokes `deliverLetter` to deliver one letter after another until all letters loaded have been delivered. Finally, the procedure `deliverLetter` makes use of `gotoRoom` to move the robot to the letter's destination and then activates *deliver* by means of a `send(fork,deliverLetter(l,color))` action.

```
proc(getLetter,[l],
         if(and(exists(colSpec,and(bel(reg(lReq(l))=colSpec,1),
                                   not(colSpec=nil))),
                bel(reg(lLoaded(l))=nil,1)),
            [gotoRoom(kwhich(reg(lOrig(l)))),
             send(fork,pickupLetter([l,kwhich(reg(lReq(l)))])),
             bel(reg(lLoaded(l))=nil,0)?],
          say("Error! Can't get letter",l))).

proc(delivAllLoaded,[],
        [say("Beginning Delivery"),
        while(exists(l,exists(c,and(bel(reg(lLoaded(l))=c,1),
                                    not(c=nil)))),
          deliverLetter(kwhich(nextDeliv)))]).

proc(deliverLetter,[l],
        if(and(exists(colSpec,and(bel(reg(lReq(l))=colSpec,1),
                                  not(colSpec=nil))),
               exists(c,and(bel(reg(lLoaded(l))=c,1),not(c=nil)))),
            [gotoRoom(kwhich(reg(lDest(l)))),
             send(fork,deliverLetter([l,kwhich(reg(lLoaded(l)))])),
             bel(reg(lLoaded(l))=nil,1)?],
          say("Error! Can't deliver letter",l))).
```

Based on the procedures `firstGet` and `firstDeliv`, we can specify the overall high-level plan, which is implemented by the procedure `main`. The body of `main` consists of a while loop which only becomes final if no letter request remains to be achieved. What happens within this while-loop depends on whether or not there still exists a letter which is to be delivered and which has not yet been picked up by the robot. If there are no such letters, then all letters loaded are delivered, and the overall plan ends.

On the other hand, if there still are letters which have to be delivered and have not yet been loaded, `main` makes use of the two procedures `firstGet` and `firstDeliv` to deliberate over the two possible behaviors: first pickup another letter, or first deliver the letters loaded. For

this purpose, it determines the expected utility `uGet` of `firstGet` by means of the epistemic test `eu` (cf. Section 8.3.1), and compares it with the expected utility `uDeliv` of `firstDeliv`. Here, we simply identify the utility of a situation with its starting time multiplied with -1, which means that an execution trace that ends earlier has a higher utility. If `uGet` is at least as high as `uDeliv`, then `main` picks up the next letter; else, it delivers all letters loaded. Thereafter, the overall plan goes in the next loop.

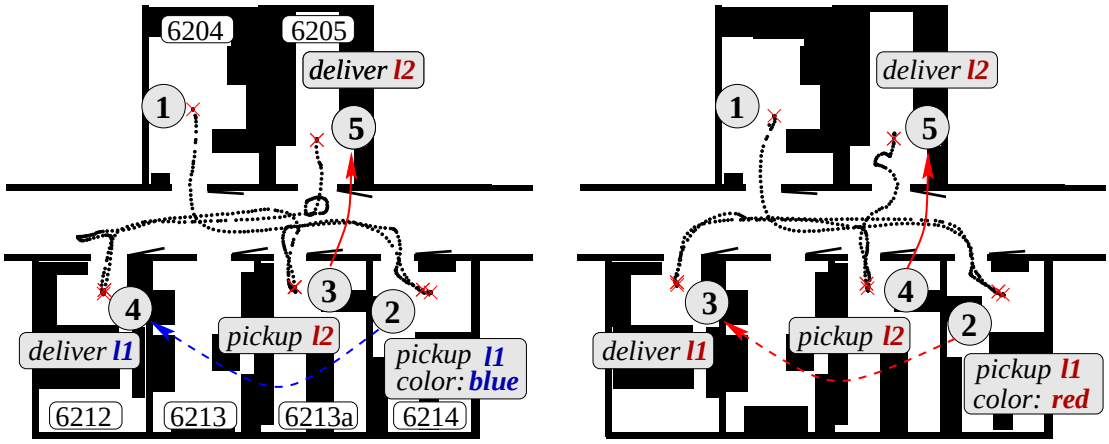
```
proc(main,[],
   while(exists(l,exists(c,and(bel(reg(lReq(l))=c,1),
                               not(c=nil)))),
      if(not(exists(l,and(exists(c,and(bel(reg(lReq(l))=c,1),
                                       not(c=nil))),
                          bel(reg(lLoaded(l))=nil,1)))),
         [say("No more letters to pickup."),
          delivAllLoaded],
        [if(exists(uGet,exists(uDeliv,
              and(and(eu(firstGet,uGet),eu(firstDeliv,uDeliv)),
                      uGet>=uDeliv))),
            [say("I will first pickup the next letter."),
             getLetter(kwhich(nextPickup))],
           [say("I will first deliver the letters."),
            delivAllLoaded])])))).
```

8.4.4 Experimental Results

Let us now consider some on-line execution traces resulting from the execution of the high-level program `main`. In all scenarios, the robot is initially in Room 6204. In the first scenario, the robot is to deliver two letters, `l1` and `l2`. Letter `l1` is to be delivered from Room 6214 to Room 6212, and Letter `l2` from Room 6213a to Room 6205. While the request for Letter `l2` specifies that the letter will be within a red envelope, the color of the envelope for Letter `l1` is unspecified. Figure 8.11 illustrates two possible executions traces of `main` in this scenario. Here, dotted lines are used to represent deliveries whose color was not specified in advance, while solid lines are used to represent deliveries resulting from a fully specified request. The labeled nodes indicate subsequent locations of the robot.

Let us first consider the execution trace illustrated in the left sub-figure. After its activation, `main` first causes the robot to travel to a pickup place, namely to Room 6214. Here, the robot asks for Letter `l1`, and is told that `l1` is within a blue envelope, that is the user presses the blue button. The robot now estimates the expected utility of the two procedures `firstGet` and `firstDeliv`. The trajectory resulting from the execution of `firstGet` is slightly shorter than that resulting from the execution of `firstDeliv`. Furthermore, as Letter `l1` is within a blue envelope and the next letter to be picked up, Letter `l2`, has been specified to be in a red envelope, the execution of `firstGet` should not result in a delay due to an ambiguous delivery. Thus, `firstGet` has a lower estimated duration, that is a higher expected utility, and the robot first travels to Room 6213a where it picks up Letter `l2`. Finally, it subsequently travels to the Rooms 6212 and 6205 to deliver the letters loaded.

Next, let us consider the execution trace illustrated in the right sub-figure. Again, the robot first travels to Room 6214 to pick up Letter `l1`. This time, however, it is told that `l1` is



(a) Letter 1 is Blue

(b) Letter 1 is Red

Figure 8.11: Colored Envelopes - Scenario I

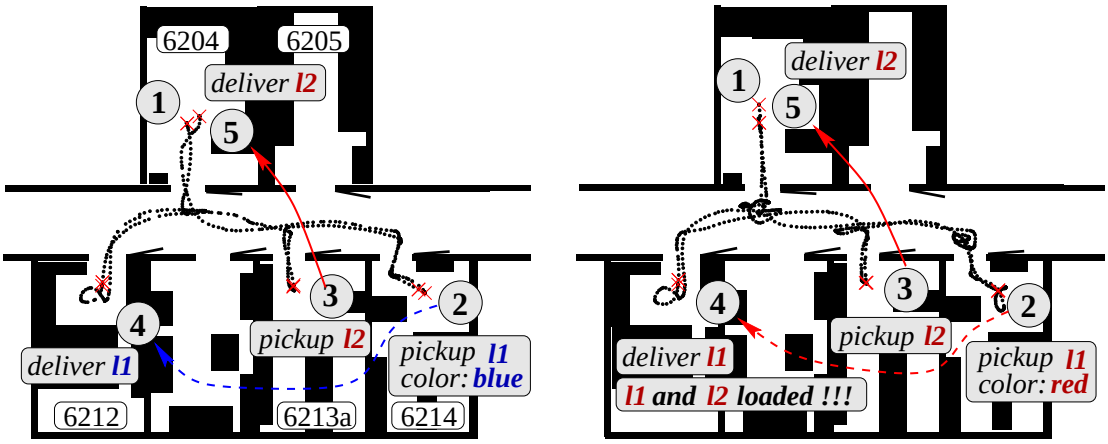
within a *red* envelope, which means that if the robot would first pick up the next letter this would result in two red envelopes being loaded. The projection of `firstGet` thus predicts a possible delay due to an ambiguous letter delivery, and as a consequence the expected utility of `firstGet` in the actual situation drops below that of `firstDeliv`. Thus, the robot first delivers Letter `l1` in Room 6212, then travels back to Room 6213a to pick up Letter `l2`, and finally delivers Letter `l2` in Room 6205.

The above scenario shows that our high-level program `main` will sometimes choose a longer trajectory in order to avoid carrying two letters with the same color. However, `main` will only avoid carrying two envelopes with the same color as long as this does not result in a major detour, as shown by our next scenario. Here, the robot is to deliver Letter `l2` to Room 6204 (not to Room 6205); otherwise this scenario corresponds to the previous one. As Room 6204 is closer to Room 6212 than Room 6205 but farther away from Room 6213a, the trajectory resulting from the execution of `firstDeliv` (the robot being in Room 6214) is significantly longer than that resulting from the execution of `firstGet`. In fact, `firstDeliv` would cause the robot to first travel to Room 6212, then back to Room 6213a and finally to Room 6204, a major detour. Accordingly, the execution of `main` causes the robot to first pick up both letters even if this results in an ambiguous delivery, as illustrated in Figure 8.12.

As final example, we used a somewhat more complex scenario where the robot is to deliver four letters. The exact delivery requests are as follows:

- Deliver Letter `l1` from Room 6214 to Room 6212, color unspecified;
- Deliver Letter `l2` from Room 6213a to Room 6205, color red;
- Deliver Letter `l3` from Room 6205 to Room 6204, color green;
- Deliver Letter `l4` from Room 6213 to Room 6204, color unspecified.

Figure 8.13 illustrates an execution trace of our high-level program `main` in this scenario. The robot first moves to Room 6205, where it picks up Letter `l3`, which is inside a green



(a) Letter 1 is Blue

(b) Letter 1 is Red

Figure 8.12: Colored Envelopes - Scenario II

envelope (node 2). Thereafter, it estimates the utility of `firstGet` and `firstDeliv`, where Letter `l1` is the next letter to be picked up. As the color of Letter `l1` is unspecified, there is a 25% probability that the execution of `firstGet` will result in the two green letters being loaded. Nevertheless, the expected utility of `firstGet` is higher than that of `firstDeliv`, so the robot travels to Room 6214 to pick up Letter `l1`. Here, the robot is given a a red letter (node 3).

At this point, the robot has loaded two letters, a green and a red one. The two letters have to be delivered to Room 6204 respectively to Room 6212. The next pickup place is in Room 6213a, where the robot could pick up Letter `l2`, which is inside a red envelope. Although Room 6213a is half on the way to Room 6212, the robot ignores the opportunity to pick up Letter `l2` because it has already loaded a red letter. Instead, it delivers the Letters `l1` and `l3` to Room 6212 (node 4) respectively to Room 6204 (node 5).

Back in Room 6204, the robot decides to move to Room 6213a to pick up Letter `l2`, which is inside a red envelope (node 6). Here, it determine that first delivering Letter `l2` in Room 6205 and then delivering Letter `l4` (whose color is unspecified) from Room 6213 to Room 6204 has a higher utility than first picking up Letter `l4` and then delivering the two letters loaded. Thus, the robot subsequently travels to Room 6205 (node 7) to deliver Letter `l2`, and to Room 6213 where it is given Letter `l4` in a blue envelope (node 8). Finally, it travels to Room 6204 where it delivers Letter `l4` (node 9) and completes its journey. We remark that the resulting overall trajectory represents a quite reasonable schedule.

The Performance of our PROLOG Interpreter We end this section with some remarks regarding the performance of our PROLOG implementation. The execution trace illustrated in the left sub-figure of Figure 8.11 consists of 502 actions, the execution trace in the right sub-figure of 569 actions. The execution traces illustrated in Figure 8.12 consist of 556 respectively of 693 actions. Finally, the execution trace illustrated in Figure 8.13 consists of 1118 actions.[9]

[9]The actual number of actions heavily depends of the actual execution trace. In particular, the number *ccUpdate* actions depends on the actual duration of a low-level navigation task and on the time the user needs

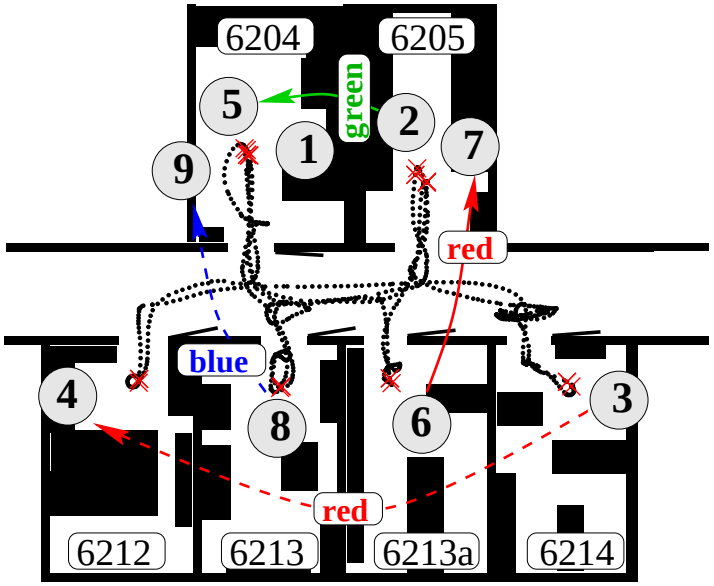


Figure 8.13: Colored Envelopes - Scenario III

In the execution trace illustrated in Figure 8.11 (left), our interpreter was able to evaluate each `eu` condition in less than 1 second (on a Pentium III 500Mhz Linux Workstation). In the execution trace illustrated in Figure 8.11 (right), the calculation of the expected utility of `firstGet` required 1.7 seconds; this additional computation time is due to the fact that the projection of `firstGet` involves an ambiguous delivery, which results in a distribution over the projected duration of the delivery. The execution traces illustrated in Figure 8.12 required similar computation time.

As for the longer execution trace illustrated in Figure 8.13, the longest computation time required to determine the expected utility of a sub-plan during its execution arose while the robot was in Room 6213a (node 6). Here, the runtime involved in the calculation of the expected utility of `firstGet` amounted to almost 7.5 seconds, in a situation consisting of over 740 actions. We remark that during these computations our interpreter determined the value of the epistemic functional fluents `kwhich(nextPickup)` and `kwhich(nextDeliv)` without considering the epistemic fluent `p`, which is possible because `nextPickup` and `nextDeliv` are defined in terms of *reg*.

8.5 Discussion

In this chapter, we have presented a prototype PROLOG implementation of the formal framework presented in the previous four chapters. In particular, we have presented a cc-Golog and a pGOLOG interpreter, and have used them to compute several example reasoning tasks employed in the previous chapters. It turned out that although the formal specification of cc-Golog and pGOLOG requires first and even second-order logic, it is relatively straightforward to write a cc-Golog respectively a pGOLOG interpreter in PROLOG. Furthermore, the implementation is relatively efficient. Except for the projection of truly belief-based programs, we

to acknowledge a receipt.

were able to compute all example reasoning tasks in at the very most a few seconds.

We also used our interpreters to control a real robot, the RWI B21 robot CARL. To do so, we coupled our PROLOG implementation to the low-level execution system BeeSoft, a state-of-the-art basic-task execution system, which has successfully been used in several real-world applications. In particular, we made use of the software package HLI which provides a uniform interface between BeeSoft and PROLOG and has already successfully been used as link between the BeeSoft system and plain GOLOG. We used the resulting run-time systems for cc-Golog and pGOLOG in several example delivery tasks. In particular, we did not only consider the on-line execution of simple non-deliberative plans, but also of plans whose outcome is conditioned on the predicted effects of different possible sub-plans.

We end this chapter with some remarks on the computational limitations of our prototype implementation regarding real-world applications. Concerning the use of cc-Golog, our experimental results suggest that our PROLOG implementation is well suited for even larger domains than those considered here. Regarding pGOLOG, however, the computation-time involved in the execution of bGOLOG plans was sometimes at the limit of being tolerable. We believe that these experiments give a good impression of the size of the problems which can be handled by our pGOLOG implementation. In particular, we remark that (at least at the moment) realistic examples like the colored letter delivery can only be handled if we restrict ourselves to bGOLOG plans which primarily make appeal to the directly observable fluent *reg*, and only occasionally make use of epistemic tests like eu. As a final anecdote, we remark that we had to abandon the development of an earlier version of a bGOLOG controller for the colored letter scenario because it was computationally much too demanding. This approach was based on the idea to represent under-specified delivery requests by a probability distribution over fully-specified requests, which required all bGOLOG plans depending on the letters to be delivered to appeal to robot's beliefs, instead of simply testing the value of some registers.

Chapter 9

Conclusions

In this chapter, we provide a summary of the logic-based framework for robot control presented in this thesis, discuss the main contributions of our approach and point out some possible future work.

9.1 Summary

While the original situation calculus which underlies GOLOG can only represent discrete change, in mobile robot applications the world changes in a continuous way, for example when the robot moves to its destination. To deal in a natural way with continuously changing properties of the world, like for example the robot's position, we have proposed a new temporal version of the situation calculus. In particular, our temporal situation calculus provides the concept of *continuous fluents*, fluents whose value change continuously over the duration of a situation. The notion of time of our temporal situation calculus is based on the idea that time advances only while the robot is waiting (via the special action *waitFor*) for the occurrence of conditions appealing to continuous fluents.

Based on this temporal situation calculus, we have developed a dialect of GOLOG, which we call cc-Golog, and which allows the specification of event-driven high-level robot plans, like for example a plan which interrupts a delivery *immediately* if the voltage level falls dangerously low. Such event-driven actions can be specified in a natural way using the new *waitFor* instruction. The semantics of ConGolog and of cc-Golog differ in that if two programs are executed concurrently, then in cc-Golog the decision which branch may execute first is not only based on the priority of the different branches but also on the execution time of actions, a concept not present in ConGolog. As a result, cc-Golog is well suited for the specification of high-level controllers where high-priority policies monitor certain condition (like a low voltage level) without blocking the execution of concurrent programs (like a letter delivery). On the other hand, cc-Golog provides a new construct, withCtrl, which allows to specify that parts of the plan shall run mutually exclusive. Using withCtrl, it is possible to specify, for example, that a high-priority policy for re-charging the batteries shall block the low-priority letter delivery.

To facilitate the actual execution of cc-Golog plans on a real robot, we have employed a layered robot control architecture where a high-level controller running cc-Golog plans is coupled to a state-of-the-art basic-task execution system like the BeeSoft system. The communication between the high-level controller and *low-level processes* like a navigation process

provided by the underlying basic-task execution system is achieved via messages. In our situation calculus framework, these messages are represented by the special primitive actions *send* and *reply*. The resulting robot control architecture is well suited to deal with processes like a navigation process whose termination does not lie under the control of the high-level controller, and which results in continuous trajectories which may differ from the predicted trajectories.

To allow both the projection and the actual (on-line) execution of the same cc-Golog plans, we have modeled every part of the basic-task execution system in our situation calculus framework. In particular, we have modeled the low-level processes by cc-Golog procedures. Having an explicit model of the low-level processes allows us to either execute a cc-Golog plan (ignoring the model of the low-level processes) or to project its effects. Based on the ability to both project and execute the same cc-Golog plan, we have introduced a local lookahead construct which allows time-bound projection during execution under user control. The use of projection during execution is quite useful in specifying intelligent robot behavior, for example to check at execution time if a robot which has to keep an appointment still has enough time to accomplish some other task before.

Besides considering issues related to continuous change and event-driven actions, we have investigated the possibility to extend the GOLOG framework to deal with probabilistic domains. In particular, to represent probabilistic beliefs we have characterized the robot's epistemic state as a distribution over situations considered possible, following [BHL95, BHL99]. Based on this representation, we have introduced belief-based bGOLOG programs, GOLOG-style programs whose tests and conditionals appeal to the robot's *beliefs*, as opposed to ordinary GOLOG programs which appeal to the value of fluents. Intuitively, the reason why bGOLOG plans may not directly appeal to the value of fluents is that in a probabilistic setting the robot is uncertain about their values – it has only beliefs about their possible values.

To represent and reason about low-level processes with *probabilistic effects*, we have introduced a probabilistic derivative of GOLOG called pGOLOG. The language pGOLOG allows the characterization of noisy low-level processes as probabilistic programs, with the idea that different probabilistic branches of the programs correspond to different outcomes of the processes. This approach does not only allow the representation of low-level processes which have probabilistic effects on the state of the world, but also of what we call *sensor processes*, low-level processes which provide (noisy) sensor information. Using pGOLOG procedures characterizing the low-level processes, we have extended GOLOG's concept of plan projection to probabilistic domains, arriving at a notion of *probabilistic projection*. Unlike ordinary projection, probabilistic projection is not concerned with which facts hold after the execution of a plan, but with the *probability* that certain facts hold after the execution of a bGOLOG plan. We have also discussed how probabilistic projection is related to the expected utility of a plan.

Besides using pGOLOG in the formalization of probabilistic projection, we employed it to characterize how the robot's probabilistic beliefs change when it executes actions (like activating a low-level process) or receives sensor input. Our approach to belief update is based on the idea to represent different states of execution of a low-level process by different pGOLOG programs. Intuitively, if a low-level program has executed some actions, then in the next situation it is characterized by what remains of the pGOLOG program after execution of these actions. In particular, we have extended our representation of the robot's epistemic state by associating with every situation considered possible a pGOLOG program modeling the actual state of execution of the low-level processes, arriving at a distribution over *configurations*.

Using this extended characterization, we can take into account that actions of the high-level controller activate low-level processes, which will subsequently cause change in the world. On the other hand, sensor inputs are used to remove configurations from the epistemic state which are not compatible with the observations.

Within this framework, we have isolated a certain class of beliefs, namely beliefs regarding what we call *directly observable* fluents. Directly observable fluents are such that the underlying situation calculus theory implies that the robot always has perfect information about them - in particular, the robot has always perfect information about messages it has received from the low-level processes. Directly observable fluents play an important role in our framework because they provide a means to efficiently derive the robot's beliefs regarding certain properties. This allows the efficient execution of bGOLOG plans which only appeal to the robot's beliefs regarding directly observable fluents.

Finally, we have extended the idea to interleave on-line execution and projection of cc-Golog plans to our probabilistic dialect bGOLOG. In particular, we have defined a local lookahead construct which allows the specification of plans appealing to probabilistic projection during execution. Similarly, we have defined a test which makes it possible to determine the expected utility of a candidate sub-plan with respect to the actual situation. Using these tests, it is possible to specify plans which compare the expected utility of different candidate sub-plans *at execution time*, and which thereafter execute the sub-plan with the highest expected utility.

We have shown several important properties of our formalization, and have implemented a cc-Golog and a pGOLOG interpreter in PROLOG. Furthermore, we have coupled our interpreters to the BeeSoft basic-task execution system, and have used the resulting run-time system to control the RWI B21 robot CARL in several experiments.

9.2 Discussion and Future Work

The extensions of the language GOLOG proposed in this thesis represent an important step towards more realistic logic-based robot controllers. Our framework provides several features which so far where only present in non-logic-based approaches: a layered control architecture which allows the interaction with low-level processes involving continuous time and change, and whose exact duration and behavior varies from time to time; a new construct which allows the specification of event-driven actions; and facilities which support the specification of mutually exclusive sub-plans. At the same time, our framework has a declarative semantics and relies on a well-understood formalism for reasoning about action and change. This gives the user a clear understanding of the effects of programs, gives her the possibility to project candidate plans based on a (probabilistic) model and allows her to investigate properties of plans, which is problematic in non-logic-based approaches.

Given that the extended GOLOG framework provides most of the features which so far were only found in non-logic-based robot programming languages, virtually every high-level robot controller written in a non-logic-based robot programming language like RPL could as well be specified using a GOLOG derivative like cc-Golog. However, in practice GOLOG will probably only become an attractive alternative to the predominating non-logic-based languages if it can really benefit from its special features in concrete applications, in particular from its ability to automatically project candidate plans both during development and at run-time. Actually, in most of today's real-world robotics applications the robots perform relatively simple high-level

tasks, like guiding people, delivering letters or serving coffee. In such scenarios, the use of logic-based robot controllers with built-in projection mechanisms may not be too convincing because projections do not seem really beneficial or even necessary, especially at run-time. We believe that it is only once robots will engage in more complex tasks that the advantages of projections will become apparent.

As an example, consider a multi-robot delivery scenario where a user makes a request to have a letter delivered by one of the robots. Then in order to determine which robot should deliver the letter, each robot might use projection to determine the cost that would arise if it were to carry out this job, and the task could then be assigned to the robot with a minimal cost estimate. On-the-fly projection becomes even more important when robots need to coordinate their activities. Suppose that two robots agree to meet somewhere at a certain time in the future. Until then they would probably want to carry out as much of their other duties as possible. Since tasks may not be interruptible at arbitrary times, each robot would be well-advised to check how much of the current task can be completed before heading off to the meeting point. To do so, some form of projection seems necessary.

However, until such complex applications are attacked the choice which high-level programming language to use is more likely to be made depending on the simplicity of the robot programming language, on the availability of a sophisticated debugging environment and on the portability and efficiency of the implementation. Unfortunately, today's existing debugging tools for GOLOG are quite restricted. As for the performance of our prototypical implementations, in particular when applied to real-world applications, the experiments in Chapter 8 demonstrate that our cc-Golog implementation is well suited for the execution of complex plans in realistic domains. On the other hand, the experiments regarding our pGOLOG implementation, like the colored letter delivery example in Section 8.4.1, suggest that our prototype implementation is only suitable for programs which are almost exclusively limited to belief tests appealing to directly observable fluents. As it seems, bGOLOG programs appealing to real-valued beliefs and to probabilistic projection tests can only be handled effectively in large probabilistic domains if we significantly improve the performance of our implementation. A promising approach would be to use some kind of database progression [LR97] during on-line execution.

Technically, the approach might also benefit from choosing a slightly different underlying formalism for reasoning about action and change. The transition semantics inherited from ConGolog requires that programs as well as formulas and fluents which are allowed to appear within programs are first-order terms. As a result, every relational fluent is represented by both an ordinary situation calculus predicate symbol and by a function symbol which denotes its reified variant. One possibility to overcome this redundancy and to reduce the amount of technical material needed in the encoding of programs as terms (cf. Appendix A) would be to rely on a formalism where fluents are considered as first-order citizens, like [PR93] or [Thi99c]. In this context, it would also be interesting to investigate whether the occurrence of self-referencing programs can be excluded without appealing to several different logical sorts distinguishing different kinds of programs.

Another issue is the applicability of our extended GOLOG dialects to (classical) planning. Here, it becomes apparent that the extensions of GOLOG presented in this thesis come at a price. Maybe the most important restriction of our framework compared with the original GOLOG is that we had to remove all nondeterministic instructions from the language. This clearly represents a significant loss of expressiveness. As an – admittedly incomplete – replacement for the nondeterministic instructions, we have introduced facilities for time-bound

projection and probabilistic projection under user control during on-line execution. Although these features clearly do not compensate the loss in expressiveness, from a practical point of view they also have advantages. In particular, if logic-based robot controllers are to be used to control robots in highly dynamic environments, like for example in the context of RoboCup, they often must come up with quick decisions. In such applications, the use of unrestricted nondeterminism is prohibitive as it potentially results in an enormous amount of reasoning, and the user writing a nondeterministic program often has no good intuition as to where significant reasoning is required. While the incremental GOLOG interpreter proposed in [dGL99b] provides features to control the amount of reasoning involved in the execution of a plan, the explicit use of projection tests provides the user with even finer-grained control.

Finally, as discussed in Section 7.3.1 there are still open issues regarding the integration of continuous change, on-line execution and probabilistic belief update. In particular, while in our approach only *reply* actions can sharpen the robot's beliefs, in domains involving continuous change it seems suggestive to also make use of the actual course of continuous trajectories to determine whether a situation is compatible with the actual state of the world. For example, if a robot faces another agent on a path involving two lanes, then it is suggestive to consider the actual trajectory covered by the other agent in order to determine which of the two lanes it probably intends to use.

Nevertheless, even though much remains to be done we believe that the work reported in this thesis represents a significant step towards more realistic logic-based robot controllers.

Index

Appendix A

Reification of Programs as Terms

In this section, we will describe how programs can be encoded as first-order terms. Furthermore, we will provide a formal account of the terms $\alpha[s]$, $\phi[s]$ and $\tau[s,t]$ informally introduced in the Sections 3.2.1 and 4.1.4.

This appendix is essentially an extension of [dGLL00], Appendix A. In particular, Sections A.1.1 to A.1.3, which develop an encoding of formulas, correspond (almost literally) to the Sections A.1 to A.3 of [dGLL00], with the only difference that here we also account for the new sorts *Real*, *Time* and *Prob*. Section A.1.4 then presents an encoding of continuous fluents and *t-form*s, concepts which were not present in [dGLL00]. The following Sections A.2 and A.3, which present an encoding of cc-Golog programs respectively of pGOLOG and bGOLOG programs as terms, are based on Section A.5 of [dGLL00] which describes an encoding of ConGolog programs as terms. However, our Sections A.2 and A.3 are significantly more complex than their counterpart in [dGLL00]. This is because in Section A.2 we do not only encode (simple) cc-Golog programs, but additionally encode projection tests and cc-Golog programs involving projection tests, unlike [dGLL00] who only consider programs without projection tests; similarly, in Section A.3 we first encode pGOLOG and bGOLOG programs as terms, and thereupon encode probabilistic projection tests and bGOLOG programs with projection tests.

A.1 Preliminaries

First, we add the following new sorts to the situation calculus: *Idx*, *PseudoSit*, *PseudoAct*, *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb*, *PseudoForm*, *PseudoCF* and *PseudoTForm*. Intuitively, elements of *Idx* denote natural numbers, and are used for building indexing functions. Elements of *PseudoAct*, *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb*, *PseudoSit*, *PseudoForm*, *PseudoCF* and *PseudoTForm* are syntactic devices to denote respectively actions, objects, reals, time points, probabilities, situations, formulas, continuous fluents and *t-form*s.

A.1.1 Sort *Idx*

We introduce the constant 0 of sort *Idx*, and a function $\mathsf{succ} : \mathit{Idx} \rightarrow \mathit{Idx}$. For them we enforce the following unique names axioms:

$$\mathsf{succ}(i) \neq 0$$
$$\mathsf{succ}(i) = \mathsf{succ}(i') \supset i = i'.$$

We define the predicate Idx of sort *Idx* as:

$$\mathsf{Idx}(i) \equiv \forall X.[... \supset X(i)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$X(0)$$
$$X(i) \supset X(\mathsf{succ}(i)).$$

Finally we assume the following domain closure axiom for sort *Idx*:

$$\forall i.\mathsf{Idx}(i)$$

where i denotes a variable of sort *Idx*.

A.1.2 Sorts *PseudoSit*, *PseudoObj*, *PseudoAct*, *PseudoReal*, *PseudoTime* and *PseudoProb*

The languages of *PseudoSit*, *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb* and *PseudoAct* are as follows:

- A constant *now* of sort *PseudoSit*;
- A function $\mathsf{nameOf}_{Sort} : Sort \rightarrow PseudoSort$ for $Sort = Obj, Act, Real, Time, Prob$;
- A function $\mathsf{var}_{Sort} : Idx \rightarrow PseudoSort$ for $Sort = Obj, Act, Real, Time, Prob$. We call terms of the form $\mathsf{var}_{Sort}(i)$ *pseudo-variables* and we use the notation z_i (or just $\mathsf{x}, \mathsf{y}, \mathsf{z}$) to denote $\mathsf{var}_{Sort}(i)$, leaving *Sort* implicit;
- A function $\mathsf{f} : PseudoSort_1 \times ... \times PseudoSort_n \rightarrow PseudoSort_{n+1}$ for each fluent or non-fluent function f of sort $Sort_1 \times ... \times Sort_n \rightarrow Sort_{n+1}$ with $Sort_i = Obj$, *Act*, *Real*, *Time*, *Prob*, *Sit* in the original language (note that if $n = 0$ then f is a constant). Note also that this does not include the epistemic fluents p (cf. Section 3.3), *pll* (cf. Section 7.1.2), *Lookahead* (cf. Section 5.2.2), *Bel* (cf. Section 6.2.1), *PBel* and *EU* (cf. Section 7.3.2).

We define the predicates $\mathsf{PseudoSit}$ of sort *PseudoSit*, $\mathsf{PseudoObj}$ of sort *PseudoObj*, $\mathsf{PseudoReal}$ of sort *PseudoReal*, $\mathsf{PseudoTime}$ of sort *PseudoTime*, $\mathsf{PseudoProb}$ of sort *PseudoProb* and $\mathsf{PseudoAct}$ of sort *PseudoAct* respectively as:

$$\begin{aligned}
\mathsf{PseudoSit}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Sit}(x)] \\
\mathsf{PseudoObj}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Obj}(x)] \\
\mathsf{PseudoReal}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Real}(x)] \\
\mathsf{PseudoTime}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Time}(x)] \\
\mathsf{PseudoProb}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Prob}(x)] \\
\mathsf{PseudoAct}(x) &\equiv \forall P_{Sit}, P_{Obj}, P_{Act}, P_{Real}, P_{Time}, P_{Prob}.[... \supset P_{Act}(x)]
\end{aligned}$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{aligned}
P_{Sit}(now) & \\
P_{Sort}(\mathsf{nameOf}_{Sort}(x)) & \quad \text{for } Sort = Obj, Act, Real, Time, Prob \\
P_{Sort}(\mathsf{z}_i) & \quad \text{for } Sort = Obj, Act, Real, Time, Prob \\
P_{Sort}(x_1) \wedge ... \wedge P_{Sort}(x_n) \quad & \supset \quad P_{Sort}(\mathsf{f}(x_1, ..., x_n)) \text{ (for each } \mathsf{f}).
\end{aligned}$$

We assume the following domain closure axioms for the sorts *PseudoSit*, *PseudoObj*, *PseudoAct*, *PseudoReal*, *PseudoTime* and *PseudoProb*:

$$\forall x.\mathsf{PseudoSit}(x)$$
$$\forall x.\mathsf{PseudoObj}(x)$$
$$\forall x.\mathsf{PseudoAct}(x)$$
$$\forall x.\mathsf{PseudoReal}(x)$$
$$\forall x.\mathsf{PseudoTime}(x)$$
$$\forall x.\mathsf{PseudoProb}(x).$$

We also enforce unique names axioms for them, that is, for all functions g, g' of any arity (including constants) introduced above:

$$g(x_1, ..., x_n) \neq g'(y_1, ..., y_m)$$
$$g(x_1, ..., x_n) = g'(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n.$$

Observe that the unique names axioms impose that $\mathsf{nameOf}(x) = \mathsf{nameOf}(y) \supset x = y$ but do not say anything about domain elements denoted by x and y since these are elements of *Act*, *Real*, *Time*, *Prob* or *Obj*.

Next we want to relate pseudo-situations, pseudo-objects, pseudo-reals, pseudo-time-points, pseudo-probabilities and pseudo-actions to real situations, object, reals, time points, probabilities and actions. In fact we do not want to relate all terms of sort *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb* and *PseudoAct* to real object and actions, but just the "closed" ones, i.e. those in which no pseudo variable z_i occur. To formalize the notion of "closedness", we introduce the predicate Closed of sort *PseudoSort* for *Sort* = *Sit*, *Obj*, *Act*, *Real*, *Time*, *Prob*, characterized by the following assertions:

$$\mathsf{Closed}(now)$$
$$\mathsf{Closed}(\mathsf{nameOf}(x))$$
$$\neg\mathsf{Closed}(\mathsf{z}_i)$$
$$\mathsf{Closed}(\mathsf{f}(x_1, ..., x_n)) \equiv \mathsf{Closed}(x_1) \wedge ... \wedge \mathsf{Closed}(x_n) \text{ for each } \mathsf{f}.$$

Closed terms of sort *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb* and *PseudoAct* are related to real objects, reals, time points, probabilities and actions by means of the function $\mathsf{decode} : \mathit{PseudoSort} \times \mathit{Sit} \rightarrow \mathit{Sort}$ for *Sort* = *Sit*, *Obj*, *Real*, *Time*, *Prob*, *Act*. We use the notation $x[s]$ to denote $\mathsf{decode}(x, s)$. Such a function is characterized by the following assertions:

$$\mathsf{decode}(now, s) = s$$
$$\mathsf{decode}(\mathsf{nameOf}(x), s) = x$$
$$\mathsf{decode}(\mathsf{f}(x_1, ..., x_n), s) = f(\mathsf{decode}(x_1, s), ..., \mathsf{decode}(x_n, s)) \text{ (for each } f).$$

A.1.3 Sort *PseudoForm*

Next we introduce *pseudo-formulas* used in tests. Specifically, we introduce:

- A function $\mathsf{p} : \mathit{PseudoSort}_1 \times ... \times \mathit{PseudoSort}_n \rightarrow \mathit{PseudoForm}$ for each non-fluent/fluent predicate p in the underlying situation calculus language. Note that this does not include the new predicates introduced in this section;

- A function $\mathsf{and} : \mathit{PseudoForm} \times \mathit{PseudoForm} \rightarrow \mathit{PseudoForm}$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\mathsf{and}(\rho_1, \rho_2)$;
- A function $\mathsf{not} : \mathit{PseudoForm} \rightarrow \mathit{PseudoForm}$. We use the notation $\neg\rho$ to denote $\mathsf{not}(\rho)$;
- A function $\mathsf{some}_{Sort} : \mathit{PseudoSort} \times \mathit{PseudoForm} \rightarrow \mathit{PseudoForm}$, for $\mathit{PseudoSort}$= *PseudoObj*, *PseudoReal*, *PseudoTime*, *PseudoProb* and *PseudoAct*. We use the notation $\exists \mathsf{z}_i.\rho$ to denote $\mathsf{some}(\mathsf{var}_{Sort}(i), \rho)$, leaving *Sort* implicit.

We define the predicate PseudoForm of sort *PseudoForm* as:

$$\mathsf{PseudoForm}(p) \equiv \forall P_{Form}.[... \supset P_{Form}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl} P_{Form}(\mathsf{p}(x_1, ..., x_n)) & & \text{(for each } \mathsf{p}) \\ P_{Form}(\rho_1) \wedge P_{Form}(\rho_2) & \supset & P_{Form}(\rho_1 \wedge \rho_2) \\ P_{Form}(\rho) & \supset & P_{Form}(\neg\rho) \\ P_{Form}(\rho) & \supset & P_{Form}(\exists \mathsf{z}_i.\rho). \end{array}$$

We assume the following domain closure axiom for the sort *PseudoForm*:

$$\forall\rho.\mathsf{PseudoForm}(\rho).$$

We also enforce unique names axioms for pseudo-formulas, that is, for all functions g, g' of any arity introduced above:

$$\begin{array}{l} g(x_1, ..., x_n) \neq g'(y_1, ..., y_m) \\ g(x_1, ..., x_n) = g'(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n. \end{array}$$

Next we formalize the notion of substitution. We introduce the function $\mathsf{sub} : \mathit{PseudoSort} \times \mathit{PseudoSort} \times \mathit{PseudoSort}' \rightarrow \mathit{PseudoSort}'$ for Sort = *Obj*, *Real*, *Prob*, *Time*, *Act* and Sort' = *Sit*, *Obj*, *Real*, *Prob*, *Time*, *Act*. We use the notation t^x_y to denote $\mathsf{sub}(x, y, t)$. Such a function is characterized by the following assertions:

$$\begin{array}{l} \mathit{now}^x_y = \mathit{now} \\ \mathsf{nameOf}(t)^x_y = \mathsf{nameOf}(t) \\ (\mathsf{z}_i)^{\mathsf{z}_i}_y = y \\ \mathsf{f}(t_1, ..., t_n)^x_y = \mathsf{f}((t_1)^x_y, ...(t_n)^x_y) \text{ (for each } \mathsf{f}). \end{array}$$

We extend the function sub to pseudo-formulas (as third argument) as follows:

$$\begin{array}{l} \mathsf{p}(t_1, ..., t_m)^x_y = \mathsf{p}((t_1)^x_y, ..., (t_n)^x_y) \text{ (for each } \mathsf{p}) \\ (\rho_1 \wedge \rho_2)^x_y = (\rho_1)^x_y \wedge (\rho_2)^x_y \\ (\neg\rho)^x_y = \neg(\rho)^x_y \\ (\exists \mathsf{z}_i.\rho)^{\mathsf{z}_i}_y = \exists \mathsf{z}_i.\rho \\ x \neq \mathsf{z}_i \supset (\exists \mathsf{z}_i.\rho)^x_y = \exists \mathsf{z}_i(\rho^x_y). \end{array}$$

Next we extend the predicate Closed to pseudo-formulas in a natural way:

$$\begin{array}{rcl} \mathsf{Closed}(\mathsf{p}(x_1, ..., x_n)) & \equiv & \mathsf{Closed}(x_1) \wedge ... \wedge \mathsf{Closed}(x_n) \text{ (for each } \mathsf{p}) \\ \mathsf{Closed}(\rho_1 \wedge \rho_2) & \equiv & \mathsf{Closed}(\rho_1) \wedge \mathsf{Closed}(\rho_2) \\ \mathsf{Closed}(\neg\rho) & \equiv & \mathsf{Closed}(\rho) \\ \mathsf{Closed}(\exists \mathsf{z}_i.\rho) & \equiv & \forall y.\mathsf{Closed}(\rho^{\mathsf{z}_i}_{\mathsf{nameOf}(y)}). \end{array}$$

We relate *closed* pseudo-formulas to real formulas by introducing a new predicate $\mathsf{Holds} : PseudoForm \times Sit$, characterized by the following assertions:

$$\begin{aligned}
\mathsf{Holds}(\mathsf{p}(x_1,...,x_n),s) &\equiv p(\mathsf{decode}(x_1,s),...,\mathsf{decode}(x_n,s)) \text{ (for each } \mathsf{p}) \\
\mathsf{Holds}(\rho_1 \wedge \rho_2, s) &\equiv \mathsf{Holds}(\rho_1,s) \wedge \mathsf{Holds}(\rho_2,s) \\
\mathsf{Holds}(\neg\rho, s) &\equiv \neg\mathsf{Holds}(\rho,s) \\
\mathsf{Holds}(\exists \mathsf{z}.\rho, s) &\equiv \exists y.\mathsf{Holds}(\rho^{\mathsf{z}}_{\mathsf{nameOf}(y)}, s)
\end{aligned}$$

where y in the last equation is any variable that does not appear in ρ. We use the notation $\phi[s]$ to denote $\mathsf{Holds}(\phi, s)$.

A.1.4 Sorts *PseudoCF* and *PseudoTForm*

Next we introduce new sorts for *pseudo-continuous-fluents* and *t-form*s, which are used as argument of *waitFor* actions. Specifically, for each continuous fluent *cf* of sort $Sort_1 \times ... \times Sort_n \rightarrow$ *t-function* in the original language we introduce a function

$$\mathsf{cf} : PseudoSort_1 \times ... \times PseudoSort_n \rightarrow PseudoCF.$$

We define the predicate $\mathsf{PseudoCF}$ of sort *PseudoCF* as:

$$PseudoCF(p) \equiv \forall P_{CF}.[... \supset P_{Form}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$P_{CF}(\mathsf{cf}(x_1,...,x_n)) \text{ (for each } \mathsf{cf}).$$

We assume the following domain closure axiom for the sort *PseudoCF*:

$$\forall\tau.\mathsf{PseudoCF}(\tau).$$

We also enforce unique names axioms for pseudo-continuous-fluents, that is, for all functions g, g' of any arity introduced above:

$$\begin{aligned}
&g(x_1,...,x_n) \neq g'(y_1,...,y_m) \\
&g(x_1,...,x_n) = g'(y_1,...,y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n.
\end{aligned}$$

Next, we define the language of *PseudoTForm* as follows:

- Functions $\mathsf{EQ}, \mathsf{GR} : PseudoCF \times PseudoReal \rightarrow PseudoTForm$. We use the notation $\mathsf{cf} = n$ respectively $\mathsf{cf} > n$ to denote $\mathsf{EQ}(\mathsf{cf}, n)$ respectively $\mathsf{GR}(\mathsf{cf}, n)$;
- A function $\mathsf{and} : PseudoTForm \times PseudoTForm \rightarrow PseudoTForm$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\mathsf{and}(\rho_1, \rho_2)$;
- A function $\mathsf{not} : PseudoTForm \rightarrow PseudoTForm$. We use the notation $\neg p$ to denote $\mathsf{not}(\rho)$.

We define the predicate $\mathsf{PseudoTForm}$ of sort *PseudoTForm* as:

$$\mathsf{PseudoTForm}(p) \equiv \forall P_{TForm}.[... \supset P_{TForm}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl} P_{TForm}(\mathsf{EQ}(\mathsf{cf}(x_1,...,x_n),n) & & \text{(for each cf)} \\ P_{TForm}(\mathsf{GR}(\mathsf{cf}(x_1,...,x_n),n) & & \text{(for each cf)} \\ P_{TForm}(\tau_1) \wedge P_{TForm}(\tau_2) & \supset & P_{TForm}(\tau_1 \wedge \tau_2) \\ P_{TForm}(\tau) & \supset & P_{TForm}(\neg\tau). \end{array}$$

We assume the following domain closure axiom for the sort *PseudoTForm*:

$$\forall\rho.\mathsf{PseudoTForm}(\rho).$$

We also enforce unique names axioms for *t-form*s, that is, for all functions g, g' of any arity introduced above:

$$\begin{array}{l} g(x_1,...,x_n) \neq g'(y_1,...,y_m) \\ g(x_1,...,x_n) = g'(y_1,...,y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n. \end{array}$$

We relate *t-form*s to real formulas by introducing a predicate $\mathsf{HoldsAt} : PseudoTForm \times Sit \times Time$, characterized by the following assertions:

$$\begin{array}{rcl} \mathsf{HoldsAt}(\mathsf{EQ}(\mathsf{cf}(x_1,...,x_n),n),s,t) & \equiv & val(cf(\mathsf{decode}(x_1,s),...,\mathsf{decode}(x_n,s),t) \\ & & \quad = \mathsf{decode}(n,s) \text{ (for each cf)} \\ \mathsf{HoldsAt}(\mathsf{GR}(\mathsf{cf}(x_1,...,x_n),n),s,t) & \equiv & val(cf(\mathsf{decode}(x_1,s),...,\mathsf{decode}(x_n,s),t) \\ & & \quad > \mathsf{decode}(n,s) \text{ (for each cf)} \\ \mathsf{HoldsAt}(\rho_1 \wedge \rho_2,s,t) & \equiv & \mathsf{Holds}(\rho_1,s,t) \wedge \mathsf{Holds}(\rho_2,s,t) \\ \mathsf{HoldsAt}(\neg\rho,s,t) & \equiv & \neg\mathsf{Holds}(\rho,s,t) \end{array}$$

We use the notation $\tau[s,t]$ to denote $\mathsf{HoldsAt}(\tau,s,t)$.

A.2 Encoding cc-Golog Programs

In this section, we encode cc-Golog programs as terms. In doing so, great care has to be taken to avoid defining self-referencing sentences. This is particularly true when we consider high-level programs which refer to projection tests, and hence to *programs*. In order to avoid running into self-referencing programs, we have chosen to first encode ordinary cc-Golog programs as terms, and thereafter, based on this encoding, to encode cc-Golog programs with projection tests as terms of a different sort.

In particular, we first introduce the sorts $Prog_{ccGolog}$ and $Env_{ccGolog}$ whose elements denote, respectively, ordinary cc-Golog programs without projection tests and environments, i.e. sets of $Prog_{ccGolog}$ procedure definitions. Next, we introduce the sort *PseudoProjTest* whose elements denote projection tests. Only cc-Golog programs without projection tests are allowed to occur within projection tests. Thereupon, we introduce the sorts $Prog_{ccGologPT}$ and $Env_{ccGologPT}$ whose elements denote cc-Golog programs with projection tests and $Prog_{ccGologPT}$ environments.

We remark that while our decision to encode cc-Golog programs and cc-Golog programs with projection tests as terms of different sorts clearly prevents the specification of self-referencing programs, it may be excessively cautious. The resulting encoding is quite complex, and there may be simpler ways to prevent the specification of self-referencing programs. We have opted for the following characterization because the introduction of different sorts provides a neat subdivision of the different types of programs we are dealing with and emphasize the ontological difference between them.

A.2.1 Sorts $Prog_{ccGolog}$ and $Env_{ccGolog}$

First, we introduce cc-Golog programs. Specifically, we introduce:

- A constant nil of sort $Prog_{ccGolog}$;
- A function act : $PseudoAct \rightarrow Prog_{ccGolog}$. As notation we write simply a to denote $\mathsf{act}(a)$;
- A function test : $PseudoForm \rightarrow Prog_{ccGolog}$. We use the notation $\rho?$ to denote $\mathsf{test}(\rho)$;
- A function seq : $Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$. We use the notation $[\sigma_1, \sigma_2]$ to denote $\mathsf{seq}(\sigma_1, \sigma_2)$;
- A function if : $PseudoForm \times Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$;
- A function while : $PseudoForm \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$;
- A function conc : $Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$;
- A function withCtrl : $PseudoForm \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$.

To deal with procedures we need to introduce the notion of environment together with that of program. We introduce:

- A finite number of functions P : $PseudoSort_1 \times ... \times PseudoSort_n \rightarrow Prog_{ccGolog}$, where $PseudoSort_i$ is either $PseudoObj$, $PseudoReal$, $PseudoProb$, $PseudoTime$ or $PseudoAct$. These functions are going to be used as procedure calls;
- A function proc : $Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$. This function is used to build procedure definitions and so we will force the first argument to have the form $P(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n})$, where $\mathsf{z}_1, ..., \mathsf{z}_n$ are used to denote the formal parameters of the defined procedure;
- A constant ϵ of sort $Env_{ccGolog}$, denoting the empty environment;
- A function addproc : $Env_{ccGolog} \times Prog_{ccGolog} \rightarrow Env_{ccGolog}$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta)$ to denote $\mathsf{addproc}(\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta))$;
- A function pblock : $Env_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathsf{pblock}(\mathcal{E}, \delta)$;
- A function c_call : $Env_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathsf{P}(\vec{t})]$ to denote $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$.

Next we introduce a predicate defined : $Prog_{ccGolog} \times Env_{ccGolog}$ meaning that a procedure is defined in an environment. It is specified as:

$$\mathsf{defined}(c, \mathcal{E}) \equiv \forall D.[... \supset D(c, \mathcal{E})]$$

where the ellipsis stands for the universal closure of the conjunction of

$$D(\mathsf{P}(\vec{x}), \epsilon; \mathsf{proc}(\mathsf{P}(\vec{y}), \sigma))$$
$$D(c, \mathcal{E}') \supset D(c, \mathcal{E}'; d).$$

Observe that procedures P are only defined in an environment $\mathcal{E}$, and that the parameters the procedure is applied to do not play any role in determining whether the procedure is defined.

Now we define the predicate $\mathsf{Prog}_{\mathsf{ccGolog}}$ of sort $Prog_{ccGolog}$ and the predicate $\mathsf{Env}_{\mathsf{ccGolog}}$ of sort $Env_{ccGolog}$ as:

$$\mathsf{Prog}_{\mathsf{ccGolog}}(\delta) \equiv \forall P_{Prog}, P_{Env}.[... \supset P_{Prog}(\delta)]$$
$$\mathsf{Env}_{\mathsf{ccGolog}}(\mathcal{E}) \equiv \forall P_{Prog}, P_{Env}.[... \supset P_{Env}(\delta)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl}
P_{Prog}(\mathsf{nil}) & & \\
P_{Prog}(\mathsf{act}(a)) & & (a \text{ pseudo-action}) \\
P_{Prog}(\rho?) & & (\rho \text{ pseudo-formula}) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}([\sigma_1, \sigma_2]) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{if}(\rho, \sigma_1, \sigma_2)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{while}(\rho, \sigma)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}(\sigma_1, \sigma_2)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{withCtrl}(\rho, \sigma)) \\
P_{Prog}(\mathsf{P}(x_1, ...x_n)) & & (\text{for each } \mathsf{P}) \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) & \supset & P_{Prog}(\{\mathcal{E}; \sigma\}) \\
P_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) & \supset & P_{Prog}([\mathcal{E} : \mathsf{P}(x_1, ..., x_n)]) \\
P_{Env}(\epsilon) & & \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) \wedge \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge & & \\
(\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) & \supset & P_{Env}(\mathcal{E}; \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma)).
\end{array}$$

We assume the following domain closure axioms for the sorts $Prog_{ccGolog}$ and $Env_{ccGolog}$:

$$\forall \sigma.\mathsf{Prog}_{\mathsf{ccGolog}}(\sigma)$$
$$\forall \mathcal{E}.\mathsf{Env}_{\mathsf{ccGolog}}(\mathcal{E}).$$

We also enforce unique names axioms for programs and environments, that is for all functions g, g' of any arity introduced above:

$$g(x_1, ...x_n) \neq g'(y_1, ...y_m)$$
$$g(x_1, ..., x_n) = g'(y_1, ...y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n.$$

We extend the predicate Closed to $Prog_{ccGolog}$ by induction on the structure of the program terms in the obvious way so as to consider *closed*, programs in which all occurrences of pseudo-variables z_i are bound by being a formal parameter of a procedure. *Only closed programs are considered legal.*

We introduce the function $\mathsf{resolve} : Env_{ccGolog} \times Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow Prog_{ccGolog}$, to be used to associate to procedure calls the environment to be used to resolve them. Namely, given the procedure P defined in the environment $\mathcal{E}$, $\mathsf{resolve}(\mathcal{E}, \mathsf{P}(\vec{t}), \delta)$ denoted by $(\delta)^{\mathsf{P}(\vec{t})}_{[\mathcal{E}:\mathsf{P}(\vec{t})]}$, suitably replaces $\mathsf{P}(\vec{t})$ by $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$ in order to obtain static scope for procedures. It is

obvious how the function can be extended to resolve whole sets of procedure calls whose procedures are defined in the environment $\mathcal{E}$. Formally this function satisfies the following assertions:

$$
\begin{aligned}
(\mathsf{nil})^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{nil}\\
(a)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= a\\
(\rho?)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \rho?\\
([\sigma_1,\sigma_2])^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [(\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]},(\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}]\\
(\mathsf{if}(\rho,\sigma_1,\sigma_2))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{if}(\rho,(\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]},(\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]})\\
(\mathsf{while}(\rho,\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{while}(\rho,(\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]})\\
(\mathsf{conc}(\sigma_1,\sigma_2))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{conc}((\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]},(\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]})\\
(\mathsf{withCtrl}(\rho,\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{withCtrl}(\rho,(\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]})\\
(\mathsf{P}(\vec{x}))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [\mathcal{E}:\mathsf{P}(\vec{x})]\\
(\mathsf{Q}(\vec{t}))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{Q}(\vec{t}) \text{ for any procedure call } \mathsf{Q}(\vec{t}) \text{ different from } \mathsf{P}(\vec{x})\\
(\{\mathcal{E}';\delta\})^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \begin{cases} \{\mathcal{E}';\delta\} & \text{if procedure } P \text{ is (re)defined in } \mathcal{E}'\\ \{\mathcal{E}';(\delta)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}\} & \text{otherwise}\end{cases}\\
([\mathcal{E}':\mathsf{Q}(\vec{t})]^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [\mathcal{E}':\mathsf{Q}(\vec{t})] \text{ for every procedure call } \mathsf{Q}(\vec{t}) \text{ and environment } \mathcal{E}'.
\end{aligned}
$$

We extend the function sub to $Prog_{ccGolog}$ (as third argument) again by induction on the structure of program terms in the natural way without doing any substitutions into environments. sub is used for substituting formal parameters with actual parameters in contextualized procedure calls. We also introduce a function $\mathsf{c\_body} : Prog_{ccGolog} \times Env_{ccGolog} \rightarrow Prog_{ccGolog}$ to be used to return the body of the procedures. Namely, $\mathsf{c\_body}(\mathsf{P}(\vec{x}),\mathcal{E})$ returns the body of the procedure P in $\mathcal{E}$ with the formal parameters substituted by the actual parameters $\vec{x}$, and thus formalizes the term $\beta_{P\vec{x}}^{\vec{v_p}}$ used in the sections 3.2.3 and 4.2.3. Formally this function satisfies the following assertions:

$\mathsf{c\_body}(\mathsf{P}(\vec{x}),\mathcal{E};\mathsf{proc}(\mathsf{P}(\vec{y}),\delta) = \delta^{\vec{y}}_{\vec{x}}$
$\mathsf{c\_body}(\mathsf{P}(\vec{x}),\mathcal{E};\mathsf{proc}(\mathsf{Q}(\vec{y}),\delta) = \mathsf{c\_body}(\mathsf{P}(\vec{x}),\mathcal{E})$ for $\mathsf{Q} \neq \mathsf{P}$.

A.2.2 Sorts *PseudoProjTest*, $Prog_{ccGologPT}$ and $Env_{ccGologPT}$

Next we introduce the sort *PseudoProjTest* whose elements denote cc-Golog projection tests. Specifically, we introduce:

- A function Lookahead : $PseudoForm \times PseudoTime \times Prog_{ccGolog} \times Prog_{ccGolog} \rightarrow PseudoProjTest$ (recall that Lookahead takes two programs as arguments: a cc-Golog plan to be projected, and a cc-Golog program modeling the low-level processes);
- A function and : $PseudoProjTest \times PseudoProjTest \rightarrow PseudoProjTest$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\mathsf{and}(\rho_1,\rho_2)$;

- A function $\mathsf{not} : \mathit{PseudoProjTest} \rightarrow \mathit{PseudoProjTest}$. We use the notation $\neg\rho$ to denote $\mathsf{not}(\rho)$.

We remark that by this definition only cc-Golog programs without projection tests are allowed to occur within a Lookahead construct. Next, we define the predicate PseudoProjTest of sort *PseudoProjTest* as:

$$\mathsf{PseudoProjTest}(p) \equiv \forall P_{PrTst}.[... \supset P_{PrTst}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{aligned} P_{PrTst}(\mathsf{Lookahead}(\phi, t, \sigma, ll_{model})) & \\ P_{PrTst}(\rho_1) \wedge P_{PrTst}(\rho_2) & \supset P_{PrTst}(\rho_1 \wedge \rho_2) \\ P_{PrTst}(\rho) & \supset P_{PrTst}(\neg\rho). \end{aligned}$$

We assume the following domain closure axiom for the sort *PseudoProjTest*:

$$\forall\rho.\mathsf{PseudoProjTest}(\rho).$$

We also enforce unique names axioms for pseudo-projection-tests. To relate projection-tests to real formulas, we extend the predicate Holds to *PseudoProjTest*. Specifically, we add the following assertions:

$$\begin{aligned} \mathsf{Holds}(\mathsf{Lookahead}(\phi, t, \sigma, ll_{model}), s) &\equiv \mathit{Lookahead}(\phi, \mathsf{decode}(t, s), \sigma, ll_{model}, s) \\ \mathsf{Holds}(\rho_1 \wedge \rho_2, s) &\equiv \mathsf{Holds}(\rho_1, s) \wedge \mathsf{Holds}(\rho_2, s) \\ \mathsf{Holds}(\neg\rho, s) &\equiv \neg\mathsf{Holds}(\rho, s) \end{aligned}$$

where *Lookahead* is the predicate defined in Section 5.2.2 to allow the projection of a cc-Golog plan. Note that Holds does not decode the pseudo-formula ϕ, which instead is directly turned over to the predicate *Lookahead*. As discussed in Section 5.2.2, the predicate Lookahead does not evaluate the truth value of ϕ in s, but in the projected execution trace that results from the execution of the program σ in s.

Next, we define the sort $\mathit{Prog}_{ccGologPT}$ of cc-Golog programs including projection tests. $\mathit{Prog}_{ccGologPT}$ is a super-sort of $\mathit{Prog}_{ccGolog}$, meaning that every element of $\mathit{Prog}_{ccGolog}$ is a legal element of $\mathit{Prog}_{ccGologPT}$. Additionally, we introduce the following functions of sort $\mathit{Prog}_{ccGologPT}$:

- A function $\mathsf{test}_{pt} : \mathit{PseudoProjTest} \rightarrow \mathit{Prog}_{ccGologPT}$. We use the notation $\rho?$ to denote $\mathsf{test}_{pt}(\rho)$;
- A function $\mathsf{if}_{pt} : \mathit{PseudoProjTest} \cup \mathit{PseudoForm} \times \mathit{Prog}_{ccGologPT} \times \mathit{Prog}_{ccGologPT} \rightarrow \mathit{Prog}_{ccGologPT}$;
- A function $\mathsf{while}_{pt} : \mathit{PseudoProjTest} \cup \mathit{PseudoForm} \times \mathit{Prog}_{ccGologPT} \rightarrow \mathit{Prog}_{ccGologPT}$;
- A function $\mathsf{withCtrl}_{pt} : \mathit{PseudoProjTest} \cup \mathit{PseudoForm} \times \mathit{Prog}_{ccGologPT} \rightarrow \mathit{Prog}_{ccGologPT}$;
- A function $\mathsf{seq}_{pt} : \mathit{Prog}_{ccGologPT} \times \mathit{Prog}_{ccGologPT} \rightarrow \mathit{Prog}_{ccGologPT}$. We use the notation $[\sigma_1, \sigma_2]$ to denote $\mathsf{seq}_{pt}(\sigma_1, \sigma_2)$;
- A function $\mathsf{conc}_{pt} : \mathit{Prog}_{ccGologPT} \times \mathit{Prog}_{ccGologPT} \rightarrow \mathit{Prog}_{ccGologPT}$.

The first four functions are variants of the corresponding constructs of sort $Prog_{ccGolog}$ which may take a projection test instead of the pseudo-formula argument, and programs of sort $Prog_{ccGologPT}$ instead of programs of sort $Prog_{ccGolog}$. The functions seq_{pt} and conc_{pt} are variants of the functions seq and conc of sort $Prog_{ccGolog}$ which take as arguments arbitrary programs of sort $Prog_{ccGologPT}$ and not just programs of sort $Prog_{ccGolog}$. These new functions are needed because the original constructs may not be used to compose programs involving projection tests. Otherwise, however, the new constructs do not differ from seq and conc. Therefore, throughout the thesis we use the convention to not distinguish the new functions from the original functions, and for example to simply write conc instead of conc_{pt}. The meaning will be clear from the context. In particular, whenever a conc instruction has as argument a program involving a projection test, it stands for conc_{pt}.

As the new functions have the same intuitive meaning than the original functions, we require that they are assigned the same semantics in terms of *Trans* and *Final*. In particular, this means that every axiom in Chapter 4 and 5 which defines *Trans* or *Final* regarding test, if, while, withCtrl, seq, conc actually is an *axiom schema* which stands for both the original axiom and for a second variant which is obtained by textual substitution of the original construct with, respectively, test_{pt}, if_{pt}, while_{pt}, $\mathsf{withCtrl}_{pt}$, seq_{pt}, or conc_{pt}

Next, just as in the case of ordinary cc-Golog without projection tests we introduce a sort $Env_{ccGologPT}$ of environments for programs of sort $Prog_{ccGologPT}$. $Env_{ccGologPT}$ is a super-sort of $Env_{ccGolog}$. In addition to the function inherited from $Env_{ccGolog}$, we introduce the following functions:

- A finite number of functions $\mathsf{P} : PseudoSort_1 \times ... \times PseudoSort_n \rightarrow Prog_{ccGologPT}$, where $PseudoSort_i$ is either *PseudoObj*, *PseudoTime*, *PseudoProb*, *PseudoReal* or *PseudoAct*. These functions are going to be used as procedure calls;

- A function $\mathsf{proc}_{pt} : Prog_{ccGologPT} \times Prog_{ccGologPT} \rightarrow Prog_{ccGologPT}$. Similar to proc, this function is used to build procedure definitions;

- A function $\mathsf{addproc}_{pt} : Env_{ccGologPT} \times Prog_{ccGologPT} \rightarrow Env_{ccGologPT}$. As before, we will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathsf{proc}(P(\vec{z}), \delta)$ to denote $\mathsf{addproc}(\mathcal{E}; \mathsf{proc}(P(\vec{z}), \delta))$;

- A function $\mathsf{pblock}_{pt} : Env_{ccGologPT} \times Prog_{ccGologPT} \rightarrow Prog_{ccGologPT}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathsf{pblock}_{pt}(\mathcal{E}, \delta)$;

- A function $\mathsf{c\_call}_{pt} : Env_{ccGologPT} \times Prog_{ccGologPT} \rightarrow Prog_{ccGologPT}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathsf{P}(\vec{t})]$ to denote $\mathsf{c\_call}_{pt}(\mathcal{E}, \mathsf{P}(\vec{t}))$.

As with the new functions of sort $Prog_{ccGologPT}$ introduced above, these new functions of sort $Prog_{ccGologPT}$ and $Env_{ccGologPT}$ are variants of the corresponding constructs of sort $Prog_{ccGolog}$ respectively $Env_{ccGolog}$. They are needed in order to allow the use of procedures appealing to projection tests. Apart from the fact that they take programs respectively environments of the sorts $Prog_{ccGologPT}$ and $Env_{ccGologPT}$, they do not differ from the original constructs. Thus, throughout this thesis we do not distinguish the new functions from the original functions; the meaning will be clear from the context. Furthermore, to ensure that they are treated the same way during execution, we postulate that every axiom in Chapter 4 and 5 which defines *Trans* or *Final* regarding pblock respectively c_call is an *axiom schema* standing for

both the original axiom and for a second variant which is obtained by textual substitution of the original construct with, respectively, pblock_{pt} or $\mathsf{c\_call}_{pt}$.

Next, we extend the predicate $\mathsf{defined}$ to $Prog_{ccGologPT}$ as first argument in analogy to Section A.2.1. Then, we define the predicate $\mathsf{Prog}_{\mathsf{ccGologPT}}$ of sort $Prog_{ccGologPT}$ and the predicate $\mathsf{Env}_{\mathsf{ccGolog}}$ of sort $Env_{ccGolog}$ as:

$$\mathsf{Prog}_{\mathsf{ccGologPT}}(\delta) \equiv \forall P_{Prog}, P_{Env}.[... \supset P_{Prog}(\delta)]$$
$$\mathsf{Env}_{\mathsf{ccGologPT}}(\mathcal{E}) \equiv \forall P_{Prog}, P_{Env}.[... \supset P_{Env}(\delta)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl}
P_{Prog}(\mathsf{nil}) & & \\
P_{Prog}(\mathsf{act}(a)) & & (a \text{ pseudo-action}) \\
P_{Prog}(\rho?) & & (\rho \text{ pseudo-formula or projection-test}) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{seq}(\sigma_1, \sigma_2)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{seq}_{pt}(\sigma_1, \sigma_2)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{if}(\rho, \sigma_1, \sigma_2)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{if}_{pt}(\rho, \sigma_1, \sigma_2)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{while}(\rho, \sigma)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{while}_{pt}(\rho, \sigma)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}(\sigma_1, \sigma_2)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}_{pt}(\sigma_1, \sigma_2)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{withCtrl}(\rho, \sigma)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{withCtrl}_{pt}(\rho, \sigma)) \\
P_{Prog}(\mathsf{P}(x_1, ...x_n)) & & (\text{for each } \mathsf{P}) \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{pblock}(\mathcal{E}, \sigma)) \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{pblock}_{pt}(\mathcal{E}, \sigma)) \\
P_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) & \supset & P_{Prog}(\mathsf{c_call}(\mathcal{E} : \mathsf{P}(x_1, ..., x_n))) \\
P_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) & \supset & P_{Prog}(\mathsf{c_call}_{pt}(\mathcal{E} : \mathsf{P}(x_1, ..., x_n))) \\
P_{Env}(\epsilon) & & \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) \wedge \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge & & \\
\quad (\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) & \supset & P_{Env}(\mathsf{addproc}(\mathcal{E}, \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma))) \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) \wedge \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge & & \\
\quad (\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) & \supset & P_{Env}(\mathsf{addproc}_{pt}(\mathcal{E}, \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma))).
\end{array}$$

We assume the following domain closure axioms for the sorts $Prog_{ccGologPT}$ and $Env_{ccGologPT}$:

$$\forall\sigma.\mathsf{Prog}_{\mathsf{ccGologPT}}(\sigma)$$
$$\forall\mathcal{E}.\mathsf{Env}_{\mathsf{ccGologPT}}(\mathcal{E}).$$

Next, we enforce unique names axioms for the sorts $Prog_{ccGologPT}$ and $Env_{ccGologPT}$. Then, we extend the predicate Closed to $Prog_{ccGologPT}$; as before, only closed programs are considered legal. Finally, we extend the functions $\mathsf{resolve}$, sub and $\mathsf{c\_body}$ to $Prog_{ccGologPT}$. This is done in complete analogy to Section A.2.1.

A.3 Encoding pGOLOG and bGOLOG Programs

In this section, we will encode pGOLOG and bGOLOG programs as terms. As in the previous section, to avoid running into self-referencing programs we introduce different sorts of programs. In particular, we first introduce the sort $Prog_{pgologS}$, whose elements denote ordinary

pGOLOG programs. Similarly, we introduce the sorts *PseudoBBForm* and $Prog_{bGolog}$, whose elements denote, respectively, belief-based formulas and bGOLOG programs without projection tests. Besides, we introduce the sorts $Env_{pGologS}$ and Env_{bGolog} whose elements denote program environments of the different sorts of programs.

Based on the sorts $Prog_{pgologS}$ and $Prog_{bGolog}$, we introduce the sort $Prog_{pgolog}$ whose elements denote "mixed" programs built from $Prog_{pgologS}$ and $Prog_{bGolog}$ programs. This sort is needed because our definition of probabilistic projection is based on the concurrent execution of a bGOLOG plan and the pGOLOG model of the low-level processes (cf. Section 7.3.2). Thereupon, we introduce the sort *PseudoPProjTst* whose elements denote probabilistic projection tests. Finally, we introduce the sorts $Prog_{bGologPT}$ and $Env_{bGologPT}$, whose elements denote bGOLOG programs with projection tests and $Prog_{bGologPT}$ environments, respectively.

A.3.1 Sorts $Prog_{pgologS}$ and $Env_{pGologS}$

First, we introduce simple pGOLOG programs. In particular, we introduce the following constants and symbols:

- A constant nil of sort $Prog_{pgologS}$;
- A function act : $PseudoAct \rightarrow Prog_{pgologS}$. As notation we write simply a to denote $\mathsf{act}(a)$;
- A function test : $PseudoForm \rightarrow Prog_{pgologS}$. We use the notation $\rho?$ to denote $\mathsf{test}(\rho)$;
- A function seq : $Prog_{pgologS} \times Prog_{pgologS} \rightarrow Prog_{pgologS}$. We use the notation $[\sigma_1, \sigma_2]$ to denote $\mathsf{seq}(\sigma_1, \sigma_2)$;
- A function if : $PseudoForm \times Prog_{pgologS} \times Prog_{pgologS} \rightarrow Prog_{pgologS}$;
- A function conc : $Prog_{pgologS} \times Prog_{pgologS} \rightarrow Prog_{pgologS}$;
- A function withCtrl : $PseudoForm \times Prog_{pgologS} \rightarrow Prog_{pgologS}$;
- A function prob : $PseudoProb \times Prog_{pgologS} \times Prog_{pgologS} \rightarrow Prog_{pgologS}$.

As in the case of cc-Golog, we introduce a sort of program environments. Specifically, we introduce:

- A finite number of functions P : $PseudoSort_1 \times ... \times PseudoSort_n \rightarrow Prog_{pgologS}$, where $PseudoSort_i$ is either *PseudoObj*, *PseudoReal*, *PseudoProb*, *PseudoTime* or *PseudoAct*. These functions are going to be used as procedure calls;
- A function proc : $Prog_{pgologS} \times Prog_{pgologS} \rightarrow Prog_{pgologS}$. This function is used to build procedure definitions and so we will force the first argument to have the form $P(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n})$, where $\mathsf{z}_1, ..., \mathsf{z}_n$ are used to denote the formal parameters of the defined procedure;
- A constant ϵ of sort $Env_{pGologS}$, denoting the empty environment;
- A function addproc : $Env_{pGologS} \times Prog_{pgologS} \rightarrow Env_{pGologS}$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta)$ to denote $\mathsf{addproc}(\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta))$;

- A function $\mathsf{pblock} : \mathit{Env}_{\mathit{pGologS}} \times \mathit{Prog}_{\mathit{pgologS}} \rightarrow \mathit{Prog}_{\mathit{pgologS}}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathsf{pblock}(\mathcal{E}, \delta)$;
- A function $\mathsf{c\_call} : \mathit{Env}_{\mathit{pGologS}} \times \mathit{Prog}_{\mathit{pgologS}} \rightarrow \mathit{Prog}_{\mathit{pgologS}}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathsf{P}(\vec{t})]$ to denote $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$.

Next, we extend the predicate $\mathsf{defined}$ to $\mathit{Prog}_{\mathit{pgologS}}$, specifying whether a simple pGOLOG procedure is defined in an $\mathit{Env}_{\mathit{pGologS}}$ environment.

$$\mathsf{defined}(c, \mathcal{E}) \equiv \forall D.[... \supset D(c, \mathcal{E})]$$

where the ellipsis stands for the universal closure of the conjunction of

$$D(\mathsf{P}(\vec{x}, \epsilon; \mathsf{proc}(\mathsf{P}(\vec{y}), \sigma))$$
$$D(c, \mathcal{E}') \supset D(c, \mathcal{E}'; d).$$

Next, we define the predicate $\mathsf{Prog_{pgologS}}$ of sort $\mathit{Prog}_{\mathit{pgologS}}$ and the predicate $\mathsf{Env_{pGologS}}$ of sort $\mathit{Env}_{\mathit{pGologS}}$ as:

$$\mathsf{Prog_{pgologS}}(\delta) \equiv \forall P_{\mathit{Prog}}, P_{\mathit{Env}}.[... \supset P_{\mathit{Prog}}(\delta)]$$
$$\mathsf{Env_{pGologS}}(\mathcal{E}) \equiv \forall P_{\mathit{Prog}}, P_{\mathit{Env}}.[... \supset P_{\mathit{Env}}(\delta)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl}
P_{\mathit{Prog}}(\mathsf{nil}) & & \\
P_{\mathit{Prog}}(\mathsf{act}(a)) & & (a \text{ pseudo-action}) \\
P_{\mathit{Prog}}(\rho?) & & (\rho \text{ pseudo-formula}) \\
P_{\mathit{Prog}}(\sigma_1) \wedge P_{\mathit{Prog}}(\sigma_2) & \supset & P_{\mathit{Prog}}([\sigma_1, \sigma_2]) \\
P_{\mathit{Prog}}(\sigma_1) \wedge P_{\mathit{Prog}}(\sigma_2) & \supset & P_{\mathit{Prog}}(\mathsf{if}(\rho, \sigma_1, \sigma_2)) \\
P_{\mathit{Prog}}(\sigma) & \supset & P_{\mathit{Prog}}(\mathsf{while}(\rho, \sigma)) \\
P_{\mathit{Prog}}(\sigma_1) \wedge P_{\mathit{Prog}}(\sigma_2) & \supset & P_{\mathit{Prog}}(\mathsf{conc}(\sigma_1, \sigma_2)) \\
P_{\mathit{Prog}}(\sigma) & \supset & P_{\mathit{Prog}}(\mathsf{withCtrl}(\rho, \sigma)) \\
P_{\mathit{Prog}}(\sigma_1) \wedge P_{\mathit{Prog}}(\sigma_2) & \supset & P_{\mathit{Prog}}(\mathsf{prob}(p, \sigma_1, \sigma_2)) \ (0 < p < 1) \\
P_{\mathit{Prog}}(\mathsf{P}(x_1, ...x_n)) & & (\text{for each } \mathsf{P}) \\
P_{\mathit{Env}}(\mathcal{E}) \wedge P_{\mathit{Prog}}(\sigma) & \supset & P_{\mathit{Prog}}(\{\mathcal{E}; \sigma\}) \\
P_{\mathit{Env}}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) & \supset & P_{\mathit{Prog}}([\mathcal{E} : \mathsf{P}(x_1, ..., x_n)]) \\
P_{\mathit{Env}}(\epsilon) & & \\
P_{\mathit{Env}}(\mathcal{E}) \wedge P_{\mathit{Prog}}(\sigma) \wedge \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge & & \\
(\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) & \supset & P_{\mathit{Env}}(\mathcal{E}; \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma)).
\end{array}$$

We assume the following domain closure axioms for the sorts $\mathit{Prog}_{\mathit{pgologS}}$ and $\mathit{Env}_{\mathit{pGologS}}$:

$$\forall \sigma.\mathsf{Prog_{pgologS}}(\sigma)$$
$$\forall \mathcal{E}.\mathsf{Env_{pGologS}}(\mathcal{E}).$$

We also enforce unique names axioms for programs and environments in the usual way. Next, we extend the predicate Closed to $\mathit{Prog}_{\mathit{pgologS}}$; as usual, only closed programs are considered legal. Then, we introduce the function $\mathsf{resolve} : \mathit{Env}_{\mathit{pGologS}} \times \mathit{Prog}_{\mathit{pgologS}} \times \mathit{Prog}_{\mathit{pgologS}} \rightarrow \mathit{Prog}_{\mathit{pgologS}}$, to be used to associate to procedure calls the environment to be used to resolve

them. Namely, given the procedure P defined in the environment $\mathcal{E}$, $\mathsf{resolve}(\mathcal{E}, \mathsf{P}(\vec{t}), \delta)$ denoted by $(\delta)^{\mathsf{P}(\vec{t})}_{[\mathcal{E}:\mathsf{P}(\vec{t})]}$, suitably replaces $\mathsf{P}(\vec{t})$ by $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$ in order to obtain static scope for procedures. Formally this function satisfies the following assertions:

$$
\begin{aligned}
(\mathsf{nil})^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{nil} \\
(a)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= a \\
(\rho?)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \rho? \\
([\sigma_1, \sigma_2])^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [(\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}, (\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}] \\
(\mathsf{if}(\rho, \sigma_1, \sigma_2))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{if}(\rho, (\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}, (\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}) \\
(\mathsf{while}(\rho, \sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{while}(\rho, (\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}) \\
(\mathsf{conc}(\sigma_1, \sigma_2))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{conc}((\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}, (\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}) \\
(\mathsf{withCtrl}(\rho, \sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{withCtrl}(\rho, (\sigma)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}) \\
(\mathsf{prob}(p, \sigma_1, \sigma_2))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{prob}(p, (\sigma_1)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}, (\sigma_2)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}) \\
(\mathsf{P}(\vec{x}))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [\mathcal{E} : \mathsf{P}(\vec{x})] \\
(\mathsf{Q}(\vec{t}))^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \mathsf{Q}(\vec{t}) \text{ for any procedure call } \mathsf{Q}(\vec{t}) \text{ different from } \mathsf{P}(\vec{x}) \\
(\{\mathcal{E}'; \delta\})^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= \begin{cases} \{\mathcal{E}'; \delta\} & \text{if procedure } P \text{ is (re)defined in } \mathcal{E}' \\ \{\mathcal{E}'; (\delta)^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]}\} & \text{otherwise} \end{cases} \\
([\mathcal{E}' : \mathsf{Q}(\vec{t})]^{\mathsf{P}(\vec{x})}_{[\mathcal{E}:\mathsf{P}(\vec{x})]} &= [\mathcal{E}' : \mathsf{Q}(\vec{t})] \text{ for every procedure call } \mathsf{Q}(\vec{t}) \text{ and environment } \mathcal{E}'.
\end{aligned}
$$

Next, we extend the functions sub and $\mathsf{c\_body}$ to $Prog_{pgologS}$ in analogy to Section A.2.1. Finally, we define the predicate $notIncludes : Prog_{pgologS} \times PseudoAct$ (cf. Section 6.2.3) as:

$$notIncludes(\delta, \alpha) \equiv \forall NI_{Prog}, NI_{Env}.[... \supset NI_{Prog}(\delta, \alpha)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$
\begin{aligned}
& NI_{Prog}(\mathsf{nil}, \alpha) && \\
& NI_{Prog}(\rho?, \alpha) && (\rho \text{ pseudo-formula}) \\
& a \neq \alpha && \supset \quad NI_{Prog}(\mathsf{act}(a), \alpha) \ (a \text{ pseudo-action}) \\
& NI_{Prog}(\sigma_1, \alpha) \wedge NI_{Prog}(\sigma_2, \alpha) && \supset \quad NI_{Prog}([\sigma_1, \sigma_2], \alpha) \\
& NI_{Prog}(\sigma_1, \alpha) \wedge NI_{Prog}(\sigma_2, \alpha) && \supset \quad NI_{Prog}(\mathsf{if}(\rho, \sigma_1, \sigma_2), \alpha) \\
& NI_{Prog}(\sigma, \alpha) && \supset \quad NI_{Prog}(\mathsf{while}(\rho, \sigma), \alpha) \\
& NI_{Prog}(\sigma_1, \alpha) \wedge NI_{Prog}(\sigma_2, \alpha) && \supset \quad NI_{Prog}(\mathsf{conc}(\sigma_1, \sigma_2), \alpha) \\
& NI_{Prog}(\sigma, \alpha) && \supset \quad NI_{Prog}(\mathsf{withCtrl}(\rho, \sigma), \alpha) \\
& NI_{Prog}(\sigma_1, \alpha) \wedge NI_{Prog}(\sigma_2, \alpha) && \supset \quad NI_{Prog}(\mathsf{prob}(p, \sigma_1, \sigma_2), \alpha) \ (p \text{ probability}) \\
& NI_{Prog}(\mathsf{P}(x_1, ...x_n), \alpha) && \text{(for each P)} \\
& NI_{Env}(\mathcal{E}, \alpha) \wedge NI_{Prog}(\sigma) && \supset \quad NI_{Prog}(\{\mathcal{E}; \sigma\}, \alpha) \\
& NI_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) && \supset \quad NI_{Prog}([\mathcal{E} : \mathsf{P}(x_1, ..., x_n)]) \\
& NI_{Env}(\epsilon, \alpha) && \\
& NI_{Env}(\mathcal{E}, \alpha) \wedge NI_{Prog}(\sigma, \alpha) \wedge && \\
& \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge (\textstyle\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) && \supset \quad NI_{Env}(\mathcal{E}; \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma), \alpha).
\end{aligned}
$$

As discussed in Section 6.2.3, we only consider pGOLOG programs which do not include *send* actions as legal models of the low-level processes. Formally, this means that every pGOLOG program σ used as a model of the low-level processes must satisfy the following:

$$\forall id, val.notIncludes(\sigma, \mathsf{nameOf}(send(id, val))).$$

A.3.2 Sorts *PseudoBBForm*, $Prog_{bGolog}$ and Env_{bGolog}

Next we introduce *belief-based-formulas* used in bGOLOG tests. Specifically, we introduce:

- A function $\mathsf{Bel\_EQ} : PseudoForm \times PseudoProb \rightarrow PseudoBBForm$. We use the notation $\mathsf{Bel}(\phi) = p$ to denote $\mathsf{Bel\_EQ}(\phi, p)$;
- A function $\mathsf{Bel\_GR} : PseudoForm \times PseudoProb \rightarrow PseudoBBForm$. We use the notation $\mathsf{Bel}(\phi) > p$ to denote $\mathsf{Bel\_GR}(\phi, p)$;
- A function $\mathsf{and} : PseudoBBForm \times PseudoBBForm \rightarrow PseudoBBForm$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\mathsf{and}(\rho_1, \rho_2)$;
- A function $\mathsf{not} : PseudoBBForm \rightarrow PseudoBBForm$. We use the notation $\neg\rho$ to denote $\mathsf{not}(\rho)$;

We define the predicate PseudoBBForm of sort *PseudoBBForm* as:

$$\mathsf{PseudoBBForm}(p) \equiv \forall P_{Form}.[... \supset P_{Form}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl} P_{Form}(\mathsf{Bel}(\phi) = p) & & \\ P_{Form}(\mathsf{Bel}(\phi) > p) & & \\ P_{Form}(\rho_1) \wedge P_{Form}(\rho_2) & \supset & P_{Form}(\rho_1 \wedge \rho_2) \\ P_{Form}(\rho) & \supset & P_{Form}(\neg\rho). \end{array}$$

We assume the following domain closure axiom for the sort *PseudoBBForm*:

$$\forall\rho.\mathsf{PseudoBBForm}(\rho).$$

We also enforce unique names axioms for pseudo-formulas, that is, for all functions g, g' of any arity introduced above:

$$\begin{array}{l} g(x_1, ..., x_n) \neq g'(y_1, ..., y_m) \\ g(x_1, ..., x_n) = g'(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n. \end{array}$$

We relate pseudo-belief-based-formulas to real formulas by extending the predicate Holds to *PseudoBBForm* (as first argument) as follows:

$$\begin{array}{rcl} \mathsf{Holds}(\mathsf{Bel}(\phi) = p, s) & \equiv & Bel(\phi, s) = \mathsf{decode}(p, s) \\ \mathsf{Holds}(\mathsf{Bel}(\phi) > p, s) & \equiv & Bel(\phi, s) > \mathsf{decode}(p, s) \\ \mathsf{Holds}(\rho_1 \wedge \rho_2, s) & \equiv & \mathsf{Holds}(\rho_1, s) \wedge \mathsf{Holds}(\rho_2, s) \\ \mathsf{Holds}(\neg\rho, s) & \equiv & \neg\mathsf{Holds}(\rho, s) \end{array}$$

where *Bel* is as defined in Section 3.3.2.

Based on the sort *PseudoBBForm*, we define the sorts $Prog_{bGolog}$ and Env_{bGolog}. Specifically, we introduce:

- A constant nil_{bb} of sort $Prog_{bGolog}$;
- A function $\mathsf{act}_{bb} : PseudoAct \rightarrow Prog_{bGolog}$. As notation we write simply a to denote $\mathsf{act}(a)$;
- A function $\mathsf{test}_{bb} : PseudoBBForm \rightarrow Prog_{bGolog}$. We use the notation $\rho?$ to denote $\mathsf{test}(\rho)$;
- A function $\mathsf{seq}_{bb} : Prog_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$. We use the notation $[\sigma_1, \sigma_2]$ to denote $\mathsf{seq}(\sigma_1, \sigma_2)$;
- A function $\mathsf{if}_{bb} : PseudoBBForm \times Prog_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$;
- A function $\mathsf{while}_{bb} : PseudoBBForm \times Prog_{bGolog} \rightarrow Prog_{bGolog}$.
- A function $\mathsf{conc}_{bb} : Prog_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$;
- A function $\mathsf{withCtrl}_{bb} : PseudoBBForm \times Prog_{bGolog} \rightarrow Prog_{bGolog}$;
- A finite number of functions $\mathsf{P} : PseudoSort_1 \times ... \times PseudoSort_n \rightarrow Prog_{bGolog}$, where $PseudoSort_i$ is either $PseudoObj$ or $PseudoAct$. These functions are going to be used as procedure calls;
- A function $\mathsf{proc}_{bb} : Prog_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$. This function is used to build procedure definitions and so we will force the first argument to have the form $P(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n})$, where $\mathsf{z}_1, ..., \mathsf{z}_n$ are used to denote the formal parameters of the defined procedure;
- A constant ϵ_{bb} of sort Env_{bGolog}, denoting the empty environment;
- A function $\mathsf{addproc}_{bb} : Env_{bGolog} \times Prog_{bGolog} \rightarrow Env_{bGolog}$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta)$ to denote $\mathsf{addproc}(\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta))$;
- A function $\mathsf{pblock}_{bb} : Env_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathsf{pblock}(\mathcal{E}, \delta)$;
- A function $\mathsf{c\_call}_{bb} : Env_{bGolog} \times Prog_{bGolog} \rightarrow Prog_{bGolog}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathsf{P}(\vec{t})]$ to denote $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$.

As with cc-Golog programs with projection tests, we use the convention to leave out the subscript $_{bb}$; it always becomes clear from the context whether a program is a simple pGOLOG programs or a bGOLOG plan. Furthermore, to ensure that the above functions are treated the same way as the corresponding constructs in simple pGOLOG programs during execution, we postulate that every axiom in Chapter 6 and 7 which defines *transPr* or *Final* regarding one pGOLOG's constructs is an *axiom schema* standing for the original axiom, a second variant which is obtained by textual substitution of the original construct with the construct with subscript $_{bb}$ – and additional variants to be discussed below.

Next, we extend the predicates defined and Closed to $Prog_{bGolog}$. Thereafter, we require domain closure and unique names axioms for the sorts $Prog_{bGolog}$ and Env_{bGolog} in the usual way. Similarly, we extend the functions resolve, sub and c_body to $Prog_{bGolog}$ in the usual way.

Finally, we extend the predicate *notIncludes* to sort $Prog_{bGolog}$. We only consider a bGOLOG program as a legal high-level plans if it only involves *send* actions (cf. Section 6.2.4). Formally, this means that a high-level plan σ must satisfy the following:

$$[\neg\exists id, val.a = send(id, val)] \supset notIncludes(\sigma, \mathsf{nameOf}(a)).$$

Additionally, we introduce the predicate $notIncludesArg : Prog_{bGolog} \times PseudoSort$ for $Sort = Obj, Act, Real, Time, Prob$. Intuitively, $notIncludesArg(\sigma, y)$ holds if the program σ does not appeal to y as argument in primitive actions, tests or procedure calls. Before we formally define *notIncludesArg*, we first define the predicate $occursIn : PseudoSort \times PseudoSort$ for $Sort = Obj, Act, Real, Time, Prob$. Intuitively, $occursIn(y, a)$ holds if the term y occurs within the term a. *occursIn* is defined as follows:

$$occursIn(y, a) \equiv \forall OI.[... \supset OI(y, a)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl} OI(\mathsf{f}(\vec{x}), \mathsf{f}(\vec{x})) & & \text{(for each pseudo-fluent function } \mathsf{f}) \\ a = \mathsf{f}(x_1, ..., x_n) \wedge & & \\ [occursIn(y, x_1) \vee ... \vee occursIn(y, x_n)] & \supset & OI(y, a). \end{array}$$

Note that we have enforced unique names for the pseudo-sorts concerned, thus $occursIn(y, a)$ actually determines whether the term y syntactically occurs in the term a. We extend *occursIn* to pseudo-formulas (as second argument) as follows:

$$\begin{array}{l} occursIn(y, \mathsf{p}(t_1, ..., t_m)) \equiv occursIn(y, t_1) \vee ...occursIn(y, t_m) \text{ (for each } \mathsf{p}) \\ occursIn(y, (\rho_1 \wedge \rho_2)) \equiv occursIn(y, \rho_1) \vee occursIn(y, \rho_2) \\ occursIn(y, (\neg\rho)) \equiv occursIn(y, \rho) \\ occursIn(y, (\exists \mathsf{z}_i.\rho)) \equiv occursIn(y, \rho). \end{array}$$

Based on *occursIn*, we define *notIncludesArg* as follows:

$$notIncludesArg(\delta, x) \equiv \forall NIA_{Prog}, NIA_{Env}.[... \supset NIA_{Prog}(\delta, \alpha)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcl} NIA_{Prog}(\mathsf{nil}, y) & & \\ \neg occursIn(y, \rho) & \supset & NIA_{Prog}(\rho?, y) \ (\rho \text{ pseudo-formula}) \\ \neg occursIn(y, a) & \supset & NIA_{Prog}(\mathsf{act}(a), y) \ (a \text{ pseudo-action}) \\ NIA_{Prog}(\sigma_1, y) \wedge NIA_{Prog}(\sigma_2, y) & \supset & NIA_{Prog}([\sigma_1, \sigma_2], y) \\ NIA_{Prog}(\sigma_1, y) \wedge NIA_{Prog}(\sigma_2, y) & \supset & NIA_{Prog}(\mathsf{if}(\rho, \sigma_1, \sigma_2), y) \\ NIA_{Prog}(\sigma, y) & \supset & NIA_{Prog}(\mathsf{while}(\rho, \sigma), y) \\ NIA_{Prog}(\sigma_1, y) \wedge NIA_{Prog}(\sigma_2, y) & \supset & NIA_{Prog}(\mathsf{conc}(\sigma_1, \sigma_2), y) \\ NIA_{Prog}(\sigma, y) & \supset & NIA_{Prog}(\mathsf{withCtrl}(\rho, \sigma), y) \\ NIA_{Prog}(\sigma_1, y) \wedge NIA_{Prog}(\sigma_2, y) & \supset & NIA_{Prog}(\mathsf{prob}(p, \sigma_1, \sigma_2), y) \\ \neg[occursIn(y, x_1) \vee ... \vee occursIn(y, x_n)] & \supset & NIA_{Prog}(\mathsf{P}(x_1, ...x_n), y) \text{ (for each } \mathsf{P}) \\ NIA_{Env}(\mathcal{E}, y) \wedge NIA_{Prog}(\sigma) & \supset & NIA_{Prog}(\{\mathcal{E}; \sigma\}, y) \\ NIA_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge & & \\ \neg[occursIn(y, x_1) \vee ... \vee occursIn(y, x_n)] & \supset & NIA_{Prog}([\mathcal{E} : \mathsf{P}(x_1, ..., x_n)]) \\ NIA_{Env}(\epsilon, y) & & \\ NIA_{Env}(\mathcal{E}, y) \wedge NIA_{Prog}(\sigma, y) \wedge & & \\ \neg\mathsf{defined}(\mathsf{P}(\vec{\mathsf{z}}), \mathcal{E}) \wedge (\bigwedge_{h,k=1}^{n} \mathsf{z}_{i_h} \neq \mathsf{z}_{i_k}) & \supset & NIA_{Env}(\mathcal{E}; \mathsf{proc}(\mathsf{P}(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n}), \sigma), y). \end{array}$$

We only consider a bGOLOG program σ as a legal high-level plans if it does not appeal to functional fluents in primitive actions or procedure calls (cf. Section 6.2.4). That is, for every functional fluent f (except the epistemic fluent *Kwhich* from Section 7.3.1) we require

$$notIncludesArg(\sigma, \mathsf{nameOf}(f(\vec{x}))).$$

A.3.3 Sort $Prog_{pgolog}$

Now that we have defined simple pGOLOG programs and simple bGOLOG programs, we define the sort $Prog_{pgolog}$ of arbitrary pGOLOG programs (without projection tests). $Prog_{pgolog}$ is a super-sort of pGOLOG and bGOLOG. In addition to the symbols inherited from its sub-sorts, we introduce the following symbol of sort $Prog_{pgolog}$:

- A function $\mathsf{conc}_{mixed} : Prog_{pgologS} \times Prog_{bGolog} \rightarrow Prog_{pgolog}$.

This symbol is used to define programs which concurrently execute a pGOLOG program and a bGOLOG plan. Note that this kind of program is used in Section 7.3.2 in the definition of probabilistic projection, which is based on the concurrent execution of a bGOLOG plan and the pGOLOG model of the low-level processes. As usual, we abuse notation and use the convention to leave out the subscript *mixed*. It always becomes clear from the context whether conc, conc_{bb} or conc_{mixed} is meant. Furthermore, we postulate that every axiom in Chapter 6 and 7 which defines *transPr* or *Final* regarding conc is an *axiom schema* which also stands for a variant which is obtained by textual substitution of conc with conc_{mixed}.

Next, we extend the predicates defined and Closed to $Prog_{pgolog}$. To enforce unique names for programs, we define the predicate $\mathsf{Prog}_{\mathsf{pgolog}}$ of sort $Prog_{pgolog}$ as:

$$\mathsf{Prog}_{\mathsf{pgolog}}(\delta) \equiv \forall P_{Prog}, P_{Env}.[... \supset P_{Prog}(\delta)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$\begin{array}{rcll}
 & & P_{Prog}(\mathsf{nil}) & \\
 & & P_{Prog}(\mathsf{nil}_{bb}) & \\
 & & P_{Prog}(\mathsf{act}(a)) & (a \text{ pseudo-action}) \\
 & & P_{Prog}(\mathsf{act}_{bb}(a)) & (a \text{ pseudo-action}) \\
 & & P_{Prog}\mathsf{test}(\rho) & (\rho \text{ pseudo-formula}) \\
 & & P_{Prog}\mathsf{test}_{bb}(\rho) & (\rho \text{ belief-based-formula}) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{seq}(\sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{seq}_{bb}(\sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{if}(\rho, \sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{if}_{bb}(\rho, \sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{while}(\rho, \sigma)) & \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{while}_{bb}(\rho, \sigma)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}(\sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}_{bb}(\sigma_1, \sigma_2)) & \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{conc}_{mixed}(\sigma_1, \sigma_2)) &
\end{array}$$

$$\begin{array}{rcl}
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{withCtrl}(\rho,\sigma)) \\
P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{withCtrl}_{bb}(\rho,\sigma)) \\
P_{Prog}(\sigma_1) \wedge P_{Prog}(\sigma_2) & \supset & P_{Prog}(\mathsf{prob}(p,\sigma_1,\sigma_2)) \\
P_{Prog}(\mathsf{P}(x_1,...x_n)) & & \text{(for each P)} \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{pblock}(\mathcal{E},\sigma)) \\
P_{Env}(\mathcal{E}) \wedge P_{Prog}(\sigma) & \supset & P_{Prog}(\mathsf{pblock}_{bb}(\mathcal{E},\sigma)) \\
P_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{z}),\mathcal{E}) & \supset & P_{Prog}(\mathsf{c_call}(\mathcal{E},\mathsf{P}(x_1,...,x_n))) \\
P_{Env}(\mathcal{E}) \wedge \mathsf{defined}(\mathsf{P}(\vec{z}),\mathcal{E}) & \supset & P_{Prog}(\mathsf{c_call}_{bb}(\mathcal{E},\mathsf{P}(x_1,...,x_n))).
\end{array}$$

We assume the following domain closure axioms for the sort $Prog_{pgolog}$:

$$\forall\sigma.\mathsf{Prog}_{\mathsf{pgolog}}(\sigma)$$

We also enforce unique names axioms for programs and environments, that is for all functions g, g' of any arity introduced above:

$$g(x_1,...x_n) \neq g'(y_1,...y_m)$$
$$g(x_1,...,x_n) = g'(y_1,...y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n.$$

Note that we did not introduce $Prog_{pgolog}$ environments. Neither did we consider conditionals or sequences involving programs both of sort $Prog_{pgologS}$ and $Prog_{bGolog}$. Every program of sort $Prog_{pgolog}$ is either a program of sort $Prog_{pgologS}$, a program of sort $Prog_{bGolog}$, or a program which concurrently executes programs of sort $Prog_{pgologS}$ and $Prog_{bGolog}$.

Finally, we extend the functions resolve, sub and c_body and the predicates *notIncludes* and *notIncludesArg* to $Prog_{pgolog}$.

A.3.4 Sorts $PseudoPProjTst$, $Prog_{bGologPT}$ and $Env_{bGologPT}$

Next we introduce the sort *PseudoPProjTst* whose elements denote *probabilistic projection tests*. Specifically, we introduce:

- Functions Proj_EQ, Proj_GR : $PseudoForm \times Prog_{bGolog} \times PseudoProb \rightarrow PseudoPProjTst$. We use the notation $\mathsf{PBel}(\phi,\sigma) = p$ to denote $\mathsf{Proj\_EQ}(\phi,\sigma,p)$, and $\mathsf{PBel}(\phi,\sigma) > p$ to denote $\mathsf{Proj\_GR}(\phi,\sigma,p)$,
- Functions EU_EQ, EU_GR : $Prog_{bGolog} \times PseudoReal \rightarrow PseudoPProjTst$. We use the notation $\mathsf{EU}(\sigma) = u$ to denote $\mathsf{EU\_EQ}(\sigma,u)$, and $\mathsf{EU}(\sigma) > u$ to denote $\mathsf{EU\_EQ}(\sigma,u)$;
- A function and : $PseudoPProjTst \times PseudoPProjTst \rightarrow PseudoPProjTst$. We use the notation $\rho_1 \wedge \rho_2$ to denote $\mathsf{and}(\rho_1,\rho_2)$;
- A function not : $PseudoPProjTst \rightarrow PseudoPProjTst$. We use the notation $\neg\rho$ to denote $\mathsf{not}(\rho)$;

We define the predicate PseudoPProjTest of sort $PseudoPProjTst$ as:

$$\mathsf{PseudoPProjTest}(p) \equiv \forall P_{ProjTst}.[... \supset P_{ProjTst}(p)]$$

where the ellipsis stands for the universal closure of the conjunction of

$$
\begin{aligned}
P_{ProjTst}(\mathsf{PBel}(\phi,\sigma) = p) & \\
P_{ProjTst}(\mathsf{PBel}(\phi,\sigma) > p) & \\
P_{ProjTst}(\mathsf{EU}(\sigma) = p) & \\
P_{ProjTst}(\mathsf{EU}(\sigma) > p) & \\
P_{ProjTst}(\rho_1) \wedge P_{ProjTst}(\rho_2) & \supset P_{ProjTst}(\rho_1 \wedge \rho_2) \\
P_{ProjTst}(\rho) & \supset P_{ProjTst}(\neg\rho).
\end{aligned}
$$

We assume the following domain closure axiom for the sort *PseudoPProjTst*:

$$\forall\rho.\mathsf{PseudoPProjTest}(\rho).$$

We also enforce unique names axioms for pseudo-formulas, that is, for all functions g, g' of any arity introduced above:

$$
\begin{aligned}
& g(x_1,...,x_n) \neq g'(y_1,...,y_m) \\
& g(x_1,...,x_n) = g'(y_1,...,y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n.
\end{aligned}
$$

We relate probabilistic projection tests to real formulas by extending the predicate Holds to *PseudoPProjTst* (as first argument) as follows:

$$
\begin{aligned}
\mathsf{Holds}(\mathsf{PBel}(\phi,\sigma) = p, s) &\equiv PBel(\phi,\sigma,s) = \mathsf{decode}(p,s) \\
\mathsf{Holds}(\mathsf{PBel}(\phi,\sigma) > p, s) &\equiv PBel(\phi,\sigma,s) > \mathsf{decode}(p,s) \\
\mathsf{Holds}(\mathsf{EU}(\sigma) = u, s) &\equiv EU(\sigma,s) = \mathsf{decode}(u,s) \\
\mathsf{Holds}(\mathsf{EU}(\sigma) > u, s) &\equiv EU(\sigma,s) > \mathsf{decode}(u,s) \\
\mathsf{Holds}(\rho_1 \wedge \rho_2, s) &\equiv \mathsf{Holds}(\rho_1,s) \wedge \mathsf{Holds}(\rho_2,s) \\
\mathsf{Holds}(\neg\rho, s) &\equiv \neg\mathsf{Holds}(\rho,s).
\end{aligned}
$$

Here, *PBel* and *EU* are the predicates defined in Section 7.3.2 to allow probabilistic projection and projection of the expected utility of a bGOLOG plan σ. Note that *PBel* simulates the concurrent execution of σ and the pGOLOG model of the low-level processes, meaning that it simulates a "mixed" program of sort $Prog_{pgolog}$.

Based on the sort *PseudoPProjTst*, we define the sorts $Prog_{bGologPT}$ and $Env_{bGologPT}$. $Prog_{bGologPT}$ is a super-sort of $Prog_{bGolog}$. Similarly, $Env_{bGologPT}$ is a super-sort of Env_{bGolog}. Besides the symbols inherited from $Prog_{bGolog}$ and from Env_{bGolog}, respectively, we introduce the following symbols of sort $Prog_{bGologPT}$ and $Env_{bGologPT}$:

- A function $\mathsf{test}_{ppt} : PseudoPProjTst \rightarrow Prog_{bGologPT}$. We use the notation $\rho?$ to denote $\mathsf{test}_{ppt}(\rho)$;
- A function $\mathsf{seq}_{ppt} : Prog_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$. We use the notation $[\sigma_1, \sigma_2]$ to denote $\mathsf{seq}_{ppt}(\sigma_1, \sigma_2)$;
- A function $\mathsf{if}_{ppt} : PseudoPProjTst \cup PseudoBBForm \times Prog_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$;
- A function $\mathsf{while}_{ppt} : PseudoPProjTst \cup PseudoBBForm \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$.
- A function $\mathsf{conc}_{ppt} : Prog_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$;
- A function $\mathsf{withCtrl}_{ppt} : PseudoPProjTst \cup PseudoBBForm \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$.

- A finite number of functions $\mathsf{P} : PseudoSort_1 \times ... \times PseudoSort_n \rightarrow Prog_{bGologPT}$, where $PseudoSort_i$ is either $PseudoObj$ or $PseudoAct$. These functions are going to be used as procedure calls;
- A function $\mathsf{proc}_{ppt} : Prog_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$. This function is used to build procedure definitions and so we will force the first argument to have the form $P(\mathsf{z}_{i_1}, ..., \mathsf{z}_{i_n})$, where $\mathsf{z}_1, ..., \mathsf{z}_n$ are used to denote the formal parameters of the defined procedure;
- A function $\mathsf{addproc}_{ppt} : Env_{bGologPT} \times Prog_{bGologPT} \rightarrow Env_{bGologPT}$. We will restrict the programs allowed to appear as the second argument to procedure definitions only. We use the notation $\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta)$ to denote $\mathsf{addproc}(\mathcal{E}; \mathsf{proc}(P(\vec{\mathsf{z}}), \delta))$;
- A function $\mathsf{pblock}_{ppt} : Env_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$. We use the notation $\{\mathcal{E}; \delta\}$ to denote $\mathsf{pblock}(\mathcal{E}, \delta)$;
- A function $\mathsf{c\_call}_{ppt} : Env_{bGologPT} \times Prog_{bGologPT} \rightarrow Prog_{bGologPT}$. We will restrict the programs allowed to appear as the second argument to procedure calls only. We use the notation $[\mathcal{E} : \mathsf{P}(\vec{t})]$ to denote $\mathsf{c\_call}(\mathcal{E}, \mathsf{P}(\vec{t}))$.

As before, the reason why we have to introduce new symbols like seq_{ppt} which take arguments of sort $Prog_{bGologPT}$ is that the corresponding constructs of sort $Prog_{bGolog}$ are restricted to programs which do not involve probabilistic projection tests. As usual, we use the convention to leave out the subscript $_{ppt}$; it always becomes clear from the context whether a program is an ordinary bGOLOG plan or a bGOLOG program with projection tests. Furthermore, to ensure that the above functions are treated the same way as the corresponding constructs in simple pGOLOG programs, we postulate that every axiom in Chapter 6 and 7 which defines *transPr* or *Final* regarding one pGOLOG's constructs is an *axiom schema* which also stands for a variant which is obtained by textual substitution of the original construct with the construct with subscript $_{ppt}$. The axioms schemas stand for no other variants than those mentioned in the Sections A.3.2 to A.3.4.

Next, we extend the predicates defined and Closed to $Prog_{bGologPT}$. Analogous to Section A.2.2, we enforce domain closure axioms for the sorts $Prog_{bGologPT}$ and $Env_{bGologPT}$ which specify that every program and environment of sort $Prog_{bGologPT}$ respectively $Env_{bGologPT}$ must be built from the constructs introduced in this section and in Section A.3.2. Thereafter, we extend the functions resolve, sub and c_body to $Prog_{bGologPT}$.

Thereafter, we extend the predicates *notIncludes* and *notIncludesArg* to sort $Prog_{bGologPT}$. As before, we only consider a bGOLOG program σ as a legal high-level plan if it only involves *send* actions and does not appeal to functional fluents in primitive actions or procedure calls. Formally, this means that a high-level plan σ must satisfy the following:

$$[\neg\exists id, val.a = send(id, val)] \supset notIncludes(\sigma, \mathsf{nameOf}(a));$$
$$notIncludesArg(\sigma, \mathsf{nameOf}(f(\vec{x}))).$$

Finally, we remark that if we wish to generate probabilistic projections of programs of sort $Prog_{bGologPT}$, we have to define a new sort which is a super-sort of $Prog_{bGologPT}$ and $Prog_{pgologS}$ and which includes "mixed" programs where a plan of sort $Prog_{bGologPT}$ is executed concurrently to a program of sort $Prog_{pgologS}$. This can be done in complete analogy to Section A.3.3. It is not required, however, if we only wish to *execute* programs of sort $Prog_{bGologPT}$.

A.4 Consistency Preservation

Just as in [dGLL00], the encoding presented here preserves consistency as stated by the following theorem:

Proposition 27: *Let $\mathcal{H}$ be the axioms defining the encoding above. Then every model of an action theory $\mathcal{D}$ involving sorts Sit, Act, Obj, Real, Time and Prob can be extended to a model of $\mathcal{H} \cup \mathcal{D}$ (involving the additional sorts Idx, PseudoSit, PseudoAct, PseudoObj, PseudoReal, PseudoTime, PseudoProb, PseudoForm, PseudoCF, PseudoTForm, $Prog_{ccGolog}$, $Env_{ccGolog}$, PseudoProjTest, $Prog_{ccGologPT}$, $Env_{ccGologPT}$, $Prog_{pgologS}$, $Env_{pGologS}$, PseudoBB-Form, $Prog_{bGolog}$, Env_{bGolog}, $Prog_{pgolog}$, PseudoPProjTst, $Prog_{bGologPT}$ and $Env_{bGologPT}$).*

Proof:

The proof is analogous to the proof of [dGLL00], Theorem 9. In particular, it suffices to observe that for each new sort $\mathcal{H}$ contains:

- A second-order axiom that explicitly defines a predicate which inductively characterizes the elements of the sort.
- An axiom that closes the domain of the new sort with respect to the characterizing predicate.
- Unique names axioms that extend the interpretation of $=$ to the new sort by induction on the structure of the elements (as imposed by the characterizing axiom).
- Axioms that characterize predicates and functions, such as Closed, decode, sub, Holds, HoldsAt, *notIncludes*, *notIncludesArg*, etc., by induction on the structure of the elements of the sort.

Hence, given a model M of the action theory $\mathcal{D}$, it is straightforward to introduce domains for the new sorts that satisfy the characterizing predicate, the domain closure axioms, and the unique names axioms for the sort, by proceeding by induction on the structure of the elements forced by the characterizing predicate, and then establishing the extension of the newly defined predicates/functions for the sort. □

Appendix B

A Second-Order Specification of Summation

In this section, we describe how a summation like $\sum_{\{s':\phi[s']\}} p(s', s)$ can be specified using second order quantification. The presentation essentially follows [BHL99]. For convenience, we assume that the language includes natural numbers as a sub-sort of the reals, and that the language offers variables that range only over these sorts. Let r, r' be variables of sort real, f and g second order function variables, that is variables ranging over all functions, and m, i, and j be variables of sort natural number. Formally, then, the sum of the weights $p(s', s)$ of all situations s' that fulfill the condition ϕ is defined as follows:

$$\begin{aligned}\sum_{\{s':\phi[s']\}} p(s', s) = r \doteq\ & \forall r'.(r' < r) \equiv \\ & \exists f, g, m. \forall i, j.(i \neq j \supset g(i) \neq g(j)) \wedge (i \leq m \supset \phi[g(i)]) \\ & \quad \wedge f(0) = 0 \\ & \quad \wedge \forall i. f(i+1) = f(i) + p(g(i), s) \\ & \quad \wedge f(m) > r'.\end{aligned}$$

Basically, this formula says that $\Sigma_{\{s':\phi[now|s']\}} p(s', s)$ is equal to r if and only if for every value r', r' is less than r if and only if there exists a finite set of m situations (enumerated by the function $g(0), ...g(m)$) satisfying ϕ whose weight $p(g(i), s)$ sum to a value greater than r'. The sum of the weights is computed by the function f, that is, intuitively, $f(i) = \sum_{0 \leq j \leq i} p(g(j), s)$. Note that this definition entails that the weight of an infinite set of situations is the limit of the sum of the weights of its elements.

Similarly, one can define a summation over *tuples* of variables. For example, the value of $w(s_1, s_2, s)$ can be summed over all situations s_1, s_2 satisfying $\phi_1(s_1)$ and $\phi_2(s_1, s_2)$ as follows:

$$\begin{aligned}&\sum_{\{s_1|\phi_1(s_1)\}} \sum_{\{s_2|\phi_2(s_1,s_2)\}} w(s_1, s_2, s) = r \doteq \\ &\quad \forall r'(r' < r) \equiv \\ &\qquad \exists f, g, m. \forall i, j.(i \neq j \supset g(i) \neq g(j)) \wedge (i \leq m \supset \phi_1(g(i))) \wedge f(0) = 0 \wedge f(m) > r' \\ &\qquad\quad \wedge \forall i.[\exists r_2. f(i+1) = f(i) + r_2 \\ &\qquad\qquad \wedge \forall r_2'(r_2' < r_2) \equiv \\ &\qquad\qquad\quad \exists f_2, g_2, m_2. \forall i_2, j_2.(i_2 \neq j_2 \supset g(i_2) \neq g(j_2)) \wedge (i_2 \leq m_2 \supset \phi_2(g(i), g_2(i_2))) \\ &\qquad\qquad\qquad \wedge f_2(0) = 0 \wedge \forall i_2. f_2(i_2 + 1) = f_2(i_2) + w(g(i), g_2(i_2), s) \wedge f_2(m_2) > r_2'].\end{aligned}$$

The last three lines of the above definition compute the inner sum, r_2, which is then used instead of p in the fourth line to compute the outer sum. All summations in this thesis can be defined analogously.

Bibliography

[ACF+98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *IJRR, Special Issue on "Integrated Architectures for Robot Control and Programming"*, 1998.

[AF94] James F. Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, October 1994.

[AFH+98] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi robot cooperation in the martha project. *IEEE Robotics and Automation Magazine (Special Issue on "Robotics & Automation in the European Union")*, 1998.

[AI99] E. Amir and P. Maynard-Reid II. Logic-based subsumption architectur. In *16th Intl' Joint Conference on Artificial Intelligence (IJCAI'99)*, 1999.

[All84] J.F. Allen. A general model of action and time. *Artificial Intelligence 23, 2*, 1984.

[BBC+95] Joachim Buhmann, Wolfram Burgard, Armin B. Cremers, Dieter Fox, Thomas Hofmann, Frank Schneider, Jiannis Strikos, and Sebastian Thrun. The mobile robot Rhino. *AI Magazine*, 16(2):31–38, Summer 1995.

[BBG99] Michael Beetz, Maren Bennewitz, and Henrik Grosskreutz. Probabilistic, prediction-based schedule debugging for autonomous robot office couriers. In *Proceedings of the 23rd Annual German Conference on Advances in Artificial Intelligence (KI-99)*, volume 1701 of *LNAI*. Springer, 1999.

[BCF+00] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.

[Bee99] Michael Beetz. Structured reactive controllers: Controlling robots that perform everyday activity. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 228–235, New York, May 1–5 1999. ACM Press.

[Bel57] R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, August 1995.

[BFG+97] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.

[BFH+98] C. Baral, L. Florian, A. Hardesty, D. Morales, N. Nogueira, and T. Son. From theory to practice: the UTEP robot in AAAI 96 and 97 robot contests. In *Proc. of the second international conference on automated agents (Agents 98)*, 1998.

[BFHS96] Wolfram Burgard, Dieter Fox, Daniel Hennig, and Timo Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proc. of the Fourteenth National Conference on Artificial Intelligence*, pages 896–901, 1996.

[BG98] M. Beetz and H. Grosskreutz. Causal models of mobile service robot behavior. In *Artificial Intelligence Planning Systems*, pages 163–170, 1998.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. European Conference on Planning (ECP-99)*. Springer, 1999.

[BG00] M. Beetz and H. Grosskreutz. Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. In *Artificial Intelligence Planning Systems*, pages 42–61, 2000.

[BHL95] F. Bacchus, J.Y. Halpern, and H. Levesque. Reasoning about noisy sensors in the situation calculus. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1933–1940, 1995.

[BHL99] F. Bacchus, J.Y. Halpern, and H. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence 111(1-2)*, 1999.

[BL99] Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *5th European Conference on Planning (ECP'99)*, 1999.

[BM94] M. Beetz and D. McDermott. Improving robot plans during their execution. In Kris Hammond, editor, *Second International Conference on AI Planning Systems*, pages 3–12, Morgan Kaufmann, 1994.

[BM97] M. Beetz and D. McDermott. Expressing transformations of structured reactive plans. In *Recent Advances in AI Planning. Proceedings of the 1997 European Conference on Planning*, pages 64–76. Springer Publishers, 1997.

[BP96] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1168–1175, Menlo Park, August4–8 1996. AAAI Press / MIT Press.

[BP98] Fahiem Bacchus and Ron Petrick. Modeling an agent's incomplete knowledge during planning and execution. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 432–443, San Francisco, June 2–5 1998. Morgan Kaufmann Publishers.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, April 1986.

[Bro91] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, January 1991.

[Bro93] R. Brooks. L: A subset of common lisp. Technical report, MIT AI Lab, 1993.

[BRP01] C. Boutilier, R. Reiter, and R. Price. Symbolic dynamic programming for first-order MDPs. In *Proc. of the Intl. Joint Conference on Artificial Intelligence*, pages 690–700, 2001.

[BRST00] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI'2000*, 2000.

[BT01] Chitta Baral and Le-Hi Tuan. Reasoning about actions in a probabilistic setting. In *Fifth Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 2001)*, 2001.

[BW91] Alan Burns and Andy Wellings. *Real-time systems and their programming languages*. Addison-Wesley, 1991.

[Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[Con92] J. Connell. SSS: A hybrid architecture applied to robot navigation. In *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, 1992.

[Dav92] Ernest Davis. Semantics for tasks that can be interrupted or abandoned. In *AIPS'92: Proc. of the First International Conference on Artificial Intelligence Planning Systems*, pages 37–44, College Park, Maryland, USA, 1992.

[dGL99a] G. de Giacomo and H. Levesque. Projection using regression and sensors. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'99)*, 1999.

[dGL99b] G. de Giacomo and H.J. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[dGLL97] G. de Giacomo, Y. Lesperance, and H. J Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'97)*, 1997.

[dGLL00] Guiseppe de Giacomo, Yves Lesperance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

[dGLS01] G. de Giacomo, H. Levesque, and S. Sardina. Incremental execution of guarded theories. *ACM Transactions on Computational Logic*, (to appear), 2001.

[dGRS98] G. de Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programms. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 1998.

[DHW94] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS'94)*, 1994.

[DK89] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Inteligence*, 5(3), 1989.

[DK01] P. Doherty and J. Kvarnström. Talplanner: A temporal logic based planner. *AI Magazine*, 2001.

[DL94] P. Doherty and W. Lukaszewicz. Circumscribing features and fluents. In *First Int'l Conference on Temporal Logic, Springer Lecture Notes in AI Vol. 827*, 1994.

[FBDT99] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *In Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1999.

[FBT97] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, March 1997.

[FBT99] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research (JAIR)*, 11, 1999.

[Fed93] C. Fedor. *TCX. An interprocess communication system for building robotic architectures. Programmer's guide to version 10.xx.* Carnegie Mellon University, Pittsurgh, PA 15213, 12 1993.

[Fir87] J. Firby. An investigation into reactive planning in complex domains. In *Proc. of AAAI-87*, pages 202–206, 1987.

[Fir89] J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* Technical report 672, Yale University, Department of Computer Science, January 1989.

[Fir95] J. Firby. The RAP language manual. Animate Agent Project Working Note AAP-6, University of Chicago, 1995.

[Fis96] George S. Fishman. *Monte Carlo - Concepts, Algorithms, and Applications.* Springer, 1196.

[FLK93] R. E. Fayek, R. Liscano, and G. M. Karam. A system architecture for a mobile robot based on activities and a blackboard control unit. In Lisa Werner, Robert; O'Conner, editor, *Proceedings of the 1993 IEEE International Conference on Robotics and Automation: Volume 2*, pages 267–274, Atlanta, GE, May 1993. IEEE Computer Society Press.

[FN71] R.E. Fikes and N.J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[For84] K. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[GA99] Emmanuel Guere and Rachid Alami. A possibilistic planner that deals with non-determinism and contingency. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'99)*, 1999.

[Gat92] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, San Jose, CA, 1992.

[Gat98] Erann Gat. On three layer architectures. In D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors, *AI and Mobile Robots*. MIT/AAAI Press, 1998.

[GB98] H. Geffner and B. Bonet. High-level planning and control with incomplete information using pomdps. In *Proc. Fall AAAI Symposium on Cognitive Robotics*, 1998.

[GBFK98] J.-S. Gutmann, W. Burgard, D. Fox, and K. Konolige. An experimental comparison of localization methods. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1998.

[GI89] M. Georgeff and F. Ingrand. Decision making in an embedded reasing system. In *Proc. of the 11 th Int. Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 972–978, Detroit, MI, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080, 1988.

[GL93] M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 1993.

[GL98a] M. Gelfond and V. Lifschitz. Action languages. *Linköping Electronic Articles in Computer and Information Science*, 3(16), 1998. URL: http://www.ep.liu.se/ea/cis/1998/018/.

[GL98b] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: preliminary report. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1998.

[GL00a] H. Grosskreutz and G. Lakemeyer. cc-golog: Towards more realistic logic-based robot controllers. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2000.

[GL00b] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'2000)*, 2000.

[GL01a] H. Grosskreutz and G. Lakemeyer. Belief update in the pGOLOG framework. In *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, 2001.

[GL01b] H. Grosskreutz and G. Lakemeyer. Online-execution of ccgolog plans. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001.

[Gre69] Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–240, Washington, D. C., May 1969. William Kaufmann.

[Gro00] H. Grosskreutz. Probabilistic projection and belief update in the pgolog framework. In *Second International Cognitive Robotics Workshop*, 2000.

[GW96] Keith Golden and Daniel Weld. Representing sensing actions: The middle ground revisited. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 174–185. Morgan Kaufmann, San Francisco, California, 1996.

[Haa87] A. R. Haas. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence. Proc. of the 1987 Workshop*, 1987.

[Har96] J. Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, University of Cambridge, 1996.

[Hay79] P. Hayes. The naive physics manifesto. In Donald Michie, editor, *Expert Systems in the Micro Electronic Age*. Edinburgh University Press, 1979.

[Hay85] P. Hayes. The second naive physics manifesto. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 1–36. Ablex, Norwood, NJ, 1985.

[HBL98] D. Hähnel, W. Burgard, and G. Lakemeyer. Golex - bridging the gap between logic (golog) and a real robot. In *Proceedings of the 22st German Conference on Artificial Intelligence (KI 98)*, 1998.

[HG01] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *Workshop Notes of the IJCAI-01 Workshop on Planning with Resources*, 2001.

[Hol95] C. Holzbauer. Ofai clp(q,r) manual. Technical Report TR-95-09, Austian Research Institute for Artificial Intelligence, Vienna, 1995.

[HT96] Christoph S. Herrmann and Michael Thielscher. Reasoning about continuous processes. In B. Clancey and D. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*, pages 639–644, Portland, OR, August 1996. MIT Press.

[KBM98] D. Kortenkamp, R.P. Bonasso, and R. Murphy. *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, 1998.

[Kel96a] T. Kelley. Modeling complex systems in the situation calculus: A case study using the dagstuhl steam boiler problem. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, 1996.

[Kel96b] T. Kelley. Reasoning about physical systems with the situation calculus. In *Common Sense 96, Third Symposium on Logical Formalizations of Commonsense Reasoning*, 1996.

[KHW95] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101(1-2*, 1998.

[KMRS97] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2), 1997.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an adl subset. In *Proc. European Conference on Planning (ECP-97)*, volume 1348 of *LNAI*, pages 273–285. Springer, 1997.

[Koe98] J. Koehler. Planning under resource constraints. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'98)*, pages 489–493, 1998.

[Kon97] Kurt Konolige. Colbert: A language for reactive control in sapphira. In *Proceedings of the German Annual Conference on Artificial Intelligence (KI'97)*, volume 1303 of *LNAI*, 1997.

[Kow92] R. A. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[KS86] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, Vienna, Austria, August 1992. John Wiley & Sons.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[Kui86] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29, 1986.

[Lak96] G. Lakemeyer. Only knowing in the situation calculus. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 14–25, Morgan Kaufmann, 1996.

[Lak99] G. Lakemeyer. On sensing and off-line interpreting in golog. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*. Springer, 1999.

[Lev96] H. J. Levesque. What is planning in the presence of sensing. In *Proc. of the National Conference on Artificial Intelligence (AAAI'96)*, 1996.

[LGM98] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[Lif97] V. Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96, 1997.

[LK92] R. Liscano and R.E. Fayekand G. M. Karam. A blackboard, activity-based control architecture for a mobile platform. In *IEEE/RSJ Intelligent Robots & Systems IROS'92*, pages 333–338, 1992.

[LL98] G. Lakemeyer and H. J. Levesque. Aol: a logic of acting, sensing, knowing, and only knowing. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann, 1998.

[LN00] Y. Lesperance and H.-K. Ng. Integrating planning into reactive high-level robot programs. In *Second International Cognitive Robotics Workshop*, 2000.

[LPR98] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. URL: http://www.ep.liu.se/ea/cis/1998/018/.

[LR94] F. Lin and R. Reiter. State constraints revisited. *Journal of logic and computation*, 4(5):655–678, 1994.

[LR95] F. Lin and R. Reiter. How to progress a database II: The STRIPS connection. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 2001–2007, Montreal, Canada, 1995.

[LR97] F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[LR98] Hector J. Levesque and Raymond Reiter. High-level robotic control: beyond planning. A position paper. In *AAAI 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, 1998.

[LRL+97] Hector J. Levesque, Raymond Reiter, Yves Lesprance, Fangzhen Lin, and Richard Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[LTJ98] Y. Lesperance, K. Tam, and M. Jenkin. Reactivity in a logic-based robot programming framework. In *AAAI'98 Fall Symposium on Cognitive Robotics*, 1998.

[McC63] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University. Reprinted 1968 in Semantic Information Processing (M.Minske ed.), MIT Press, 1963.

[McC80] J. McCarthy. Circumscription – A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13, 1980.

[McD82] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6, 1982.

[McD91] D. McDermott. A reactive plan language. Research Report YALEU/DCS/RR-864, Yale University, 1991.

[McD92a] D. McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.

[McD92b] D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, www.cs.yale.edu/AI/Planning/xfrm.html, 1992.

[McD94] D. McDermott. An algorithm for probabilistic, totally-ordered temporal projection. Research Report YALEU/DCS/RR-1014, Yale University, www.cs.yale.edu/AI/Planning/xfrm.html, 1994.

[MH69] John McCarthy and P. J. Hayes. Some philosphical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, 1969.

[Mil96] Rob Miller. A case study in reasoning about actions and continuous change. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'96)*, 1996.

[ML98] Stephen M. Majercik and Michael L. Littman. Maxplan: A new approach to probabilistic planning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, 1998.

[Moo85] Robert C. Moore. A formal theory of knowledge and action. In *Formal Theories of the Commonsense World*. Ablex Publishing Corp., Norwood, New Jersey, 1985.

[MPP+01] P. Mateus, A. Pacheco, J. Pinto, A. Sernadas, and C. Sernadas. Probabilistic situation calculus. *Annals of Mathematics and Artificial Intelligence*, 2001. To appear.

[MR91] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, volume 2, pages 634–639, Anaheim, California, USA, July 1991. AAAI Press/MIT Press.

[MS96] Rob Miller and Murray Shanahan. Reasoning about discontinuities in the event calculus. In *KR'96: Principles of Knowledge Representation and Reasoning*, pages 63–74. Morgan Kaufmann, 1996.

[MT97] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1997.

[MT01] Yves Martin and Michael Thielscher. Addressing the qualification problem in flux. In F. Baader, G. Brewka, and T. Eiter, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 2174 of *LNAI*, Vienna, Austria, September 2001. Springer.

[Mye96] Karen L. Myers. A procedural knowledge approach to task-level control. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS'96)*, pages 158–165. AAAI Press, 1996.

[Nil84] Nils J. Nilsson. Shakey the robot. Technical report, SRI International, 1984.

[PBC+97] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the 1st International Conference on Autonomous Agents*, pages 253–261, New York, February 5–8 1997. ACM Press.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., 1988.

[Ped89] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, 1989.

[Pin94] Javier Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, February 1994.

[Pin97] Javier Pinto. Integrating discrete and continuous change in a logical framework. *Computational Intelligence, 14(1)*, 1997.

[Poo96] David Poole. A framework for decision-theoretic planning I: Combining the situation calculus, conditional plans, probability and utility. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI'96)*. Morgan Kaufmann Publishers, 1996.

[Poo97] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):5–56, 1997.

[Poo98] David Poole. Decision theory, the situation calculus and conditional plans. *Linköping Electronic Articles in Computer and Information Science*, 3(8), 1998. URL: http://www.ep.liu.se/ea/cis/1998/008/.

[PR93] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Proc. 10th Int. Conf. on Logic Programming*, 1993.

[PR95] J. Pinto and R. Reiter. Reasoning about time in the situation calculus. *Annals of Mathematics and Artificial Intelligence*, 14, 1995.

[PR99] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[PS92] M. Peot and D. Smith. Conditional nonlinear planning. In J. Hendler, editor, *AIPS'92: Proc. of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, San Mateo, CA, 1992. Kaufmann.

[PSSM00] J. Pinto, A. Sernadas, C. Sernadas, and P. Mateus. Non-determinism and uncertainty in the situation calculus. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 8(2):127–149, 2000.

[Put94] M. Puterman. *Markov Decision Processes*. Wiley, New York, 1994.

[PW92] J.Scott. Penberthy and Daniel.S. Weld. Ucpop: A sound, complete partial order planer for ADL. In *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, 1992.

[Rei91] Ray Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a ccompleteness result for goal regression. In *In Artificial Intelligence and Mathematic Theory of Computation: Papers in Honor of John McCarthy*, 1991.

[Rei93] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[Rei96] Ray Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 2–13, 1996.

[Rei98] R. Reiter. Sequential, temporal golog. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998.

[Rei00] R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Second International Cognitive Robotics Workshop*, 2000.

[Rei01] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence - a Modern Approach*. Prentice Hall, 1995.

[Sac75] E. D. Sacerdoti. The nonlinear nature of plans. In *Proc. of the 4th International Joint Conference on Artificial Intelligence (IJCA-75)*, 1975.

[San89a] Erik Sandewall. Combining logic and differential equations for describing real-world systems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 412–420, 1989.

[San89b] Erik Sandewall. Filter prederential entailment for the logic of action in almost continuous worlds. In *IJCAI'89*, 1989.

[San94] Erik Sandewall. *Features and Fluents*. Oxford University Press, 1994.

[Sar01] Sebastian Sardina. Local conditional high-level robot programs. In *Workshop on Nonmonotonic Reasoning, Action, and Change at IJCAI-01 (NRAC-01)*, 2001.

[SB98] Trans Cao Son and Chitta Baral. Formalizing sensing actions - a transition function based approach. In *AAAI'98 Fall Symposium on Cognitive Robotics*, pages 13–20, 1998.

[SB00] Dirk Schulz and Wolfram Burgard. Probabilistic state estimation techniques for maintaining object-based world models of mobile robots. In *Second International Cognitive Robotics Workshop*, 2000.

[Sch90] L. K. Schubert. Monotonic solution to the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, Mass., 1990.

[SGH+97a] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. Sullivan. A layered architecture for office delivery robots. In *First International Conference on Autonomous Agents (Agents'97)*, pages 235–242, 1997.

[SGH+97b] Reid G. Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, Joseph O'Sullivan, and Manuela M. Veloso. Xavier: Experience with a layered robot architecture. *ACM magazine* Intelligence, 1997.

[SGH+97c] R.G. Simmons, R. Goodwin, K.Z. Haigh, S. Koenig, J. O'Sullivan, and M.M. Veloso. Xavier: Experience with a layered robot architecture. *ACM SIGART Bulletin* Intelligence, 8 (1–4), 1997.

[Sha90] M. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'90)*, 1990.

[Sha96] Murray Shanahan. Noise and the common sense informatic situation for a mobile robot. In *Proc. of the National Conference on Artificial Intelligence (AAAI'96), Vol. 2*, pages 1098–1103, 1996.

[Sha97] M. Shanahan. Noise, non-determinism and spatial uncertainty. In *Proc. of the National Conference on Artificial Intelligence (AAAI'97)*, 1997.

[Sha98] M. Shanahan. Reinventing shakey. In *AAAI 1998 Fall Symposium on Cognitive Robotics.*, 1998.

[Sha99] Murray Shanahan. The event calculus explained. In Michael Wooldridge and Manuela M. Veloso, editors, *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer, 1999.

[Sho87] Yoav Shoham. Nonmonotonic logics: Meaning and utility. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'87)*, 1987.

[Sho88] Yoav Shoham. *Reasoning about Change.* MIT Press, 1988.

[SK95] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partially observable environments. In *Proc. International Joint Conference on Artificial Intelligence*, 1995.

[SL93] R. Scherl and H. J. Levesque. The frame problem and knowledge producing actions. In *Proc. of the National Conference on Artificial Intelligence*, pages 689–695, 1993.

[Sou01] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *Proc. of the Intl. Joint Conference on Artificial Intelligence (IJCAI'2001)*, 2001.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[SW98] David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, pages 889–896, Menlo Park, July 26–30 1998. AAAI Press.

[SW00] M. Shanahan and M. Witkowski. High-level robot control through logic. In *Proceedings ATAL 2000*, 2000.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TBB+98] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schimdt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT/AAAI Press, 1998.

[TBB+99] S. Thrun, M. Bennewitz, W. Burgard, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Minerva: A second-generation museum tour-guide robot. In *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, 1999.

[Ter94] E. Ternovskaia. Interval situation calculus. In *Proc. of the ECAI'94 Workshop W5 on Logic and Change*, 1994.

[Thi98] Michael Thielscher. Towards state update axioms: Reifying successor state axioms. In L. F. del Cerro, J. Dix, and U. Furbach, editors, *JELIA*, volume 1489 of *LNAI*, pages 248–263, Dagstuhl, Germany, 1998. Springer.

[Thi99a] Michael Thielscher. Fluent calculus planning with continuous change. *Linköping Electronic Articles in Computer and Information Science*, 4(11), 1999. URL: http://www.ep.liu.se/ea/cis/1999/011/.

[Thi99b] Michael Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299, 1999.

[Thi99c] Michael Thielscher. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 1999.

[Thi00a] Michael Thielscher. The Fluent Calculus: A Specification Language for Robots with Sensors in Nondeterministic, Concurrent, and Ramifying Environments.

Technical Report CL-2000-01, Computational Logic Group, Department of Computer Science, Dresden University of Technology, October 2000.

[Thi00b] Michael Thielscher. Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 109–120, Breckenridge, CO, April 2000. Morgan Kaufmann.

[Thi01a] Michael Thielscher. Inferring implicit state knowledge and plans with sensing actions. In F. Baader, G. Brewka, and T. Eiter, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 2174 of *LNAI*, Vienna, Austria, September 2001. Springer.

[Thi01b] Michael Thielscher. The Qualification Problem: A solution to the problem of anomalous models. *Artificial Intelligence*, 2001.

[Ver83] Steven A. Vere. Planning in time: Windows and durations for activities and goals. In *IEEE Transact. on Pattern Analysis and Machine Intelligence*, volume PAMI-5, pages 246–267. IEEE, May 1983.

[War76] D. H. D. Warren. Generating conditional plans and programs. In *Proc. of the AISB Summer Conference*, 1976.

[WAS98] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 897–904, Menlo Park, 1998. AAAI Press.

[Wol96] S. Wolfram. *The Mathematica Book*. Wolfram Media, 1996.

Curriculum Vitae

Name:	Henrik Grosskreutz
Geburtsdatum:	9.5.1972
Geburtsort:	Berlin
Schulbildung:	
1977-1980	École de la Fontaine in Niamey, Niger
1980-1983	École du Plateau in Dakar, Senegal
1983-1991	Taunusschule Königstein
1991	Abitur
Zivildienst:	
1991-1992	Zivildienst beim DRK Hochtaunus
Studium:	
1992-1994	Informatikstudium an der Universität Würzburg
1994	Vordiplom
1994-1995	Informatikstudium an der Universität Caen, Frankreich
1995	Licence d' Informatique
1995-1998	Informatikstudium an der Universität Bonn
1998	Diplom
1998-2001	Stipendiat im Graduiertenkolleg "Informatik und Technik" an der RWTH Aachen