Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-based Systems Group
Prof. Gerhard Lakemeyer, Ph. D.

Diplomarbeit

Situation Calculus-based Online Plan Recognition in Continuous Domains

Christoph Schwering

vorgelegt im Fach Informatik
im Winter 2011

First examiner:	Prof. Gerhard Lakemeyer, Ph. D.
Second examiner:	Prof. Dr. Matthias Jarke
Advisors:	Dipl.-Inform. Daniel Beck Dipl.-Inform. Stefan Schiffer
Submitted:	25. November 2011

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Requirements . . . 2
1.3 Approach . . . 3
1.4 Outline . . . 5

2 Related Work **7**
2.1 Plan Recognition . . . 7
2.1.1 First-Order Logic as Modeling Language . . . 8
2.1.2 Plan Recognition in the Situation Calculus . . . 10
2.1.3 Markov Model-based Approaches . . . 10
2.1.4 Plan Recognition as Planning . . . 11
2.2 Temporal and Spatial Modeling . . . 12
2.2.1 Qualitative Modeling . . . 12
2.2.2 Quantitative Modeling . . . 13
2.3 Automotive Scene Identification . . . 14
2.4 Summary . . . 15

3 Foundations **17**
3.1 Situation Calculus . . . 17
3.2 Golog . . . 18
3.3 Notation . . . 21

4 Modeling **23**
4.1 Passing Maneuver with Instantaneous Velocity and Yaw . . . 24
4.2 Improving Realism with Preconditions . . . 27
4.3 Passing Maneuver with Instantaneous Acceleration and Yaw . . . 28
4.4 Summary . . . 30

5 Semantics **31**
5.1 Time . . . 31
5.1.1 Temporal Sequential Golog and cc-Golog . . . 32
5.1.2 Combining Temporal Sequential Golog and cc-Golog . . . 32
5.2 Concurrency . . . 34
5.3 Nondeterminism . . . 35
5.4 Robustness . . . 36
5.4.1 Non-Probabilistic Robustness . . . 37

5.4.2 Probabilistic Extensions of the Situation Calculus and Golog . 38
5.4.3 Robustness through Stochastic Actions 39
5.4.4 Atomic Complex Actions 55
5.5 Summary 57

6 Plan Recognition 59
6.1 Plan Recognition by Satisfiability 60
6.2 Incremental Plan Recognition 61
6.3 Plan Recognition by Program Execution 62
6.3.1 Non-Incremental Plan Recognition by Program Execution . . . 63
6.3.2 Incremental Plan Recognition by Incremental Program Execution 64
6.3.3 Heuristic Incremental Plan Recognition 66
6.4 Multi-Agent Plan Recognition 68
6.5 Data Robustness 69
6.5.1 Tolerance towards Agent's Fuzziness 69
6.5.2 Sensor Noise with Stochastic Actions 73
6.6 A Simple Example 74
6.6.1 The Basic Action Theory 75
6.6.2 How the Interpreter Works 76
6.7 Summary 79

7 Evaluation 81
7.1 Modeling Revisited 81
7.2 Implementation 82
7.2.1 Interpreter 82
7.2.2 Constraint Solvers 87
7.2.3 Driving Simulation 88
7.3 Experimental Results 89
7.3.1 Passing Maneuver 90
7.3.2 Cautious versus Aggressive Passing 96
7.4 Summary 103

8 Conclusion and Future Work 105
8.1 Conclusion 105
8.2 Future Work 107

Bibliography 109

List of Figures

1.1 General structure of our approach to plan recognition. 4

4.1 Passing maneuver with instantaneous changes of velocity and yaw. . . 23
4.2 Passing maneuver of V against W. 26
4.3 Vehicle trajectory to achieve a certain yaw. 28
4.4 Dependency of time on velocity and yaw. 29

5.1 Tree of outcome situations of a stochastic Golog program. 40
5.2 Value determination of two sequential stochastic actions. 44

6.1 Visualization of $TimeSit$ macro. 60
6.2 Probability density functions. 71
6.3 Trajectories of passing maneuvers in the model and reality. 72
6.4 A very simple continuous world. 74
6.5 Execution tree of the interpreter. 78
6.6 Refined structure of our approach to plan recognition. 79

7.1 The `trans/7` predicate from the interpreter. 84
7.2 Screenshots of TORCS with plan recognition output. 88
7.3 Programs used in the passing maneuver example. 91
7.4 Plan recognition of passing maneuvers. 93
7.5 Online plan recognition of a passing maneuver. 94
7.6 Alternative maneuvers to pass a car in the presence of a third car. . . 96
7.7 Linear approximation of uniform acceleration. 98
7.8 Programs for aggressive and cautious passing maneuvers. 101
7.9 Visualization of acceleration and deceleration. 102

List of Tables

7.1 Influence of search horizon on probability and runtime. 85
7.2 Runtime of sampling on different setups. 87
7.3 Passing maneuver test results. 97
7.4 Cautious and aggressive maneuver test results. 102

Chapter 1

Introduction

1.1 Motivation

In the future, cars are expected to drive more and more autonomously. This development has been, and will be, gradually. For example, a driving assistance technology called cruise control has undergone a steady improvement. Cruise control is intended for highway traffic. Initially, it simply maintained a fixed speed set by the driver, then it was extended to slow down the vehicle when the traffic demands it. Recent variants of the system even control the steering wheel in order to keep the car in the lane.

Extrapolating this gradual development shows that it is unlikely that cars will suddenly drive autonomously flawlessly in all situations. The first areas of application will be rather simple scenarios such as highway traffic. When the autonomously driving vehicle gets into a situation it cannot control, the driver needs to be adverted to take over control of and responsibility for the car. Such situations may occur due to construction zones or reckless human drivers, for example. Since it takes some time for the human to be ready to drive the car (about three to five seconds), incontrollable situations need to be recognized in advance.

This work is concerned with the problem of *recognizing the plan* of other traffic participants. The word *plan* means a driver's intention and the actions to achieve his or her goal. If such plans could be recognized online, that is, while they are pursued and while the associated actions are being carried out, possible future situations could be predicted and they could evaluated by their risk potential.

In this thesis, typical behavior patterns of traffic participants are specified in terms of *programs* written in a programming language specifically designed for this purpose. By *program*, we mean an abstract specification of an agent's behavior. For example, such a program might state that to overtake another vehicle, a car needs to swing out, pass the other car and it finally needs to go back into the lane. In contrast, a *plan* denotes the concrete actions an agent has carried out or will carry out to achieve his or her goal.[1] However, both terms are obviously closely related as a plan is a concrete instantiation of an abstract program in a concrete situation.

As part of the trend to more sophisticated driver assistance systems, cars will be equipped with more and more sensors to percept their environment. Further, car to car communication will provide means to exchange sensor data. We assume that at least each car's position is known to the other cars.

Hence, a driver's plan is recognized if the respective program can be executed in harmony with the environment's observations. The actual plan is exactly the chosen execution of the program. Note that multiple different programs may explain the observations, and even for a single program, there may be different ways to execute it in accordance with the observations. *The goal of this work* is to define a semantics of a programming language that serves this purpose, and to show how program execution and observations can be incorporated.

The basis for the programming language to be defined is Golog (Levesque et al., 1997). Golog is a high-level robot control language that provides the modeler with great expressive power: it includes conditional statements, loops, and even nondeterministic constructs. Based on the situation calculus (McCarthy and Hayes, 1969; Reiter, 1991), Golog has a sound logical foundation. This has led to a number of dialects of Golog, which will be exploited to form a new language.

1.2 Requirements

The programs that specify how drivers behave in traffic constitute the so-called plan library. According to the convention from Section 1.1, the term should be "program library" rather than "plan library." However, we stick with "plan library" for historical reasons.

The idea is to find a set of plans, one for each vehicle, from the plan library, where each plan explains the observations made by a car using its sensors. We assume that at least the global position of each vehicle is accessible. Then, the plans could be extrapolated to determine whether or not a critical situation will occur.

There are multiple aspects of plan recognition in an automobile environment that we want to look into:

Continuous Time. Since there are many physical values involved in the automobile domain, time must be handled explicitly, that is, quantitatively. Furthermore, continuous change is required so that, for example, the position of a car at a certain point in time can be determined.

[1]An exception from this rule will be the term "plan library" in the upcoming section which simply means a predefined set of programs and therefore should rather be "program library."

While observations are associated with explicit timestamps, the modeling language should support a rather descriptive representation of time. That is, the modeler should not be forced to write “change lane at time 50, pass the other car and return to the original lane at time 100.” Instead, in the plan library, time should be implicit such as in “change lane when you are near the other car, pass it and change lane back when you are far enough ahead.”

Concurrency. There are two kinds of concurrency that need to be considered:

- Different actors act at the same time. For example, two cars might change lanes simultaneously.
- Even isolated actors do things concurrently. For example, a driver should indicate while changing the lane.

The first kind of concurrency is only used internally in the plan recognition system. The second type, however, may be used by the modeler in programs of the plan library.

Robustness. There is not one unique way to overtake a car, for instance. Therefore, the plan recognition should be robust. At least the following kinds of robustness exist:

- Time: Timing constraints should not be too fixed. Different drivers usually change lanes at different points in time and distances in order to overtake another car.
- Actions: In some cases, some actions might be left out in reality. For example, indicating during a lane change might be skipped.
- Observations: Assume that the trace of a car during a lane change is described by some function. In reality, drivers will not drive exactly on this line. If there are only few observations, then timing robustness might arrange the actions so that they explain the observations. If there are more observations than actions that should explain them, probably some kind of tolerance towards data deviation is necessary. A further source of uncertainty is sensor noise.

1.3 Approach

The expected input to our plan recognition method is a sequence of time-stamped observations. Then, the problem is to execute a program from the plan library in such a way that it explains the observations. Hence, the problem of plan recognition actually becomes a planning problem. The question is whether or not there is some way to achieve the goal, that is, explains the observations. This makes demands on the

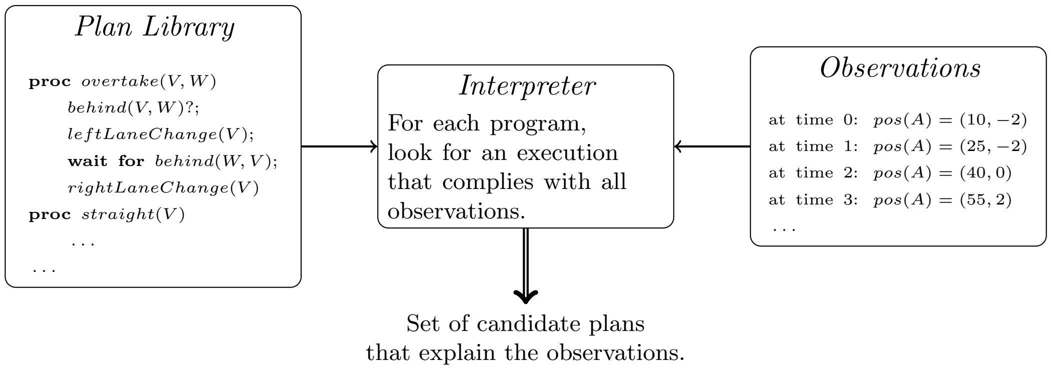

Plan Library
proc overtake(V, W)
behind(V, W)?;
leftLaneChange(V);
wait for behind(W, V);
rightLaneChange(V)
proc straight(V)
...
...
Interpreter
For each program, look for an execution that complies with all observations.
Observations
at time 0: pos(A) = (10, −2)
at time 1: pos(A) = (25, −2)
at time 2: pos(A) = (40, 0)
at time 3: pos(A) = (55, 2)
...
Set of candidate plans that explain the observations.

Figure 1.1: General structure of our approach to plan recognition. Given some candidate programs from the plan library and a sequence of observations, the interpreter's task is to search for an execution of the programs in accordance with the observations. A successful execution corresponds to a recognized plan.

robustness of the program, especially with respect to timing. The general structure is depicted in Figure 1.1.

The underlying formalism is the situation calculus (McCarthy and Hayes, 1969; Reiter, 1991), a logic language. Plans are formulated in a dialect of the situation calculus-based programming language Golog (Levesque et al., 1997). Observations are represented as first-order sentences about what holds in the world at a certain point in time.

It looks as a much simpler approach to plan recognition to completely leave out actions and define plans as sequences of logic formulas like Nagel and Arens (2005). For example, a passing maneuver could be described as one car being in the left lane and faster than another car. But with such a purely declarative way of plan recognition, extrapolation of plans appears to be impossible. This is because one only knows what would probably hold at some later point in time, but nothing about how these changes are achieved. Precise knowledge about the actions that change(d) the world and when they happen(ed) seems to be necessary to accurately recognize an agent's plan. The recognized plan should reflect reality as well as possible, because this knowledge is needed to continue the plan in simulation and check for critical situations.

Hence, one goal of this work is to bridge relatively declarative plans on the one side and concrete situation terms with actions and timestamps on the other side.

The presented approach is not specific to automotive applications. It should be applicable in other domains, particular ones which inhere physical values and/or time.

1.4 Outline

The next section gives an overview of existing work in the field of plan recognition. Chapter 3 formally introduces the situation calculus and Golog. Chapter 4 exemplifies the modeling language. The subsequent chapter defines the semantics of an extension of the situation calculus, focusing on the core aspects of plan recognition in the application domain: the handling of time, concurrency and robustness. Then, Chapter 6 discusses ways to do plan recognition by executing a Golog program that explains observations of the environment. Finally, Chapter 7 presents testing results of the proposed plan recognition framework. Chapter 8 concludes.

Chapter 2

Related Work

This chapter shortly describes some of the related work in the fields of plan recognition, temporal and spatial reasoning, and automotive-specific work.

2.1 Plan Recognition

There are three different types of plan recognition (Armentano, 2005; Carberry, 2001):

- Intended plan recognition: The user tries to suggest to the system what he is doing.
- Keyhole plan recognition: The user does not know or care that a system tries to recognize his plans.
- Obstructed plan recognition: The user tries to mislead the system.

Most approaches deal with keyhole and/or intended plan recognition. The model consists of a set of domain-specific recipes to achieve some goal, the plan library. Each recipe is some structure containing actions each of which may have a corresponding precondition and a definition of its effects (Carberry, 2001). The task of the system is to infer the agent's goal.

Generally, most work on plan recognition can be grouped in probabilistic approaches and consistency-based approaches (Armentano, 2005). Probabilistic algorithms usually use Hidden Markov Models or Bayesian Networks to rank the plans in the plan library by the probability that the plan is the user's intention. Consistency-based procedures determine all plans that do not contradict the user's actions and thus might be the user's intention.

2.1.1 First-Order Logic as Modeling Language

Kautz and Allen (1986) propose a consistency-based approach to plan recognition. The user provides an action taxonomy which defines specializations and decompositions of actions.

In logic, the specialization and decomposition relationships are defined with the binary predicate # which asserts that an action variable is an instance of an action type. $\#(e, \mathit{type})$ is also read as "a *type*-action occurred." For example, $\#(e, \mathit{maneuver})$ holds if e is an instance of a *maneuver*-action. Specialization is then expressed with axioms like

$$\forall e \,.\, \#(e, passingManeuver) \supset \#(e, maneuver)$$

which represents that *maneuver* specializes to *passingManeuver*.

Decomposition is used to define the way an action is executed. This is meant not only in a sequential way but may also impose constraints on timing. A passing maneuver might be decomposed as

$$\begin{aligned}\forall e \,.\, \#(e, passingManeuver) \supset\ & \#(S(1,e), changeLane) \wedge \\ & \#(S(2,e), pass) \wedge \\ & \#(S(3,e), changeLaneBack) \wedge \\ & startPos(S(2,e)) = endPos(S(1,e)) \wedge \\ & startPos(S(3,e)) = endPos(S(2,e)) \wedge \\ & Starts(T(S(1,e)), T(e)) \wedge \\ & During(T(S(2,e)), T(e)) \wedge \\ & Finished(T(S(3,e)), T(e)) \wedge \\ & Meets(T(S(1,e)), T(S(3,e)))\end{aligned}$$

where $S(i,e)$ is the i-th sub-action of e and $T(e)$ is the time interval at which e occurs. The temporal relationships *Starts*, *During*, *Finished* and *Meets* are taken from Allen's (1983) Interval Algebra, an interval-based logic for qualitative reasoning over time (cf. Section 2.2.1). The decomposed *passingManeuver* can still be further specialized, for example, to *illegalPassingManeuver* where *changeLane* specializes to *changeLaneToRight* (overtaking on the right lane is not allowed on many country's highways).

With circumscription (McCarthy, 1980) of #, the following assumptions are enforced:

1. The known ways of action specializations are the only ones.
2. All actions are purposeful. That is, if some action occurred and it is part of the decomposition of a bunch of complex actions, one of these complex actions occurred.

From observed actions, one can infer the underlying plans using nonmonotonic deduction. For example, *changeLane* might be a part of a maneuver to leave the highway and it might also be part of $passingManeuver$, and *pass* might be part of some third plan in addition to being part of $passingManeuver$. Then, an observation of a *changeLane* action can be explained by a passing maneuver or a leaving maneuver. An additional observation of a *pass* action reduces the set of consequences to only the $passingManeuver$ plan.

The theory lacks a concept of action effects on predicates; actions must be sensed directly. An observation could be that $\#(E_1, changeLaneToRight)$ holds. With Assumption 1, one can deduce that $\#(E_1, changeLane)$. With this knowledge and Assumption 2, one can further deduce that a passing maneuver occurs or occurred. For this passing maneuver, the above decomposition axiom gives a prediction like, for example, the overtaking car will pass the other car and then change the lane back.

A key feature of this approach to plan recognition is specialization of actions. For domains like the cooking world in (Kautz and Allen, 1986), specialization appears to be very useful: there are many kinds of meals that can be cooked, one of which is a noodle dish, and there are again many kinds of noodles and sauces. For automotive driving, specialization seems to be less useful, because action hierarchies seem to be rather flat.

According to Charniak and Goldman (1991), the approach essentially boils down to minimal set covering. Furthermore, they state that this is the wrong tool for plan recognition and abduction in general. Two common plans might be better explanation of some observations than a single very uncommon plan.

As a consequence of their criticism of Kautz and Allen (1986), Charniak and Goldman (1991) propose a probabilistic model of plan recognition.

As in (Kautz and Allen, 1986), actions are compositions of other actions. Plans are simply more complex actions. They are represented similarly to (Kautz and Allen, 1986) as decomposition entailments. The antecedent asserts that some variable is an instance of the complex action, i.e. the plan. The consequent then asserts that the subactions are again certain actions. In a somewhat adapted syntax, a plan decomposition could look like

$$\begin{aligned} Inst(e, passingManeuver) \supset\ & Inst(gotoFastLaneStep(e), changeLane) \land \\ & Inst(passStep(e), pass) \land \\ & Inst(gotoSlowLaneStep(e), changeLane) \land \\ & agent(gotoFastLaneStep(e)) = agent(passStep(e)) \land \\ & agent(passStep(e)) = agent(gotoSlowLaneStep(e)). \end{aligned}$$

Plan schemas like this one are transformed into Bayesian networks whose root nodes are the top-level plans. The random variables either represent propositions like equality constraints in the example, or instance constraints. Propositions are Boolean random variables whereas the sample space of instance constraints is the set of potential types.

2.1.2 Plan Recognition in the Situation Calculus

One plan recognition framework for ConGolog (Giacomo et al., 2000) was proposed by Goultiaeva and Lespérance (2006). ConGolog is an extension of Golog (Levesque et al., 1997), a high-level robot control programming language based on the situation calculus (McCarthy and Hayes, 1969; Reiter, 1991). The details of both, the situation calculus and Golog are explained in Section 3.1 and Section 3.2. For now, it is only relevant that ConGolog provides a way to execute a given program step-by-step. Programs usually consist of nested sequences, conditional statements, loops, and, at the lowest level, primitive actions.

The approach of Goultiaeva and Lespérance (2006) starts off with a plan library consisting of ConGolog programs. As further input, it expects a sequence of primitive actions. For such a stream of actions, a set of candidate plans is thinned out incrementally. A plan is a valid candidate if the observed action sequence is a prefix of some deduction of the plan (in grammatical terms).

Ideas from Goultiaeva and Lespérance (2006) are used in Chapter 6. However, this approach is not enough for plan recognition in highway traffic. For one, a sensor for primitive actions is unrealistic in this scenario. For another, ConGolog and as a consequence the plan recognition mechanism do not include an explicit representation of time.

2.1.3 Markov Model-based Approaches

Bui et al. (2002) model the plan library as hierarchy of a Markov Decision Processes. Markov Decision Processes can be used to model dynamic domains where actions may have nondeterministic effects, that is, an action performed in a given state leads to a certain new state with some probability. A mapping from states to actions is called a policy. At a higher level of the hierarchy, the policy chooses a lower-level sub-policy in each state. Then, the problem of plan recognition becomes essentially the problem of recognizing the agent's top-level policy.

Bui et al. (2002) approach this problem with what they call Abstract Hidden Markov Model and particle filtering. This solution mainly focuses on the uncertainty in plan recognition. They identify three sources of uncertainty: the refinement at each level in the plan hierarchy is nondeterministic and can therefore result in different sub-plans. Similarly, at the leaf-level of the hierarchy actions are nondeterministic, too, that is, they have unpredictable effects. The third source of uncertainty stems from the fact that in reality sensors are noisy.

Another probabilistic approach is proposed by Geib and Goldman (2009). They use simple yet powerful grammars represented as trees to express plans. Nodes in these trees can be AND or OR nodes which means that either all or one of the children has to be executed, respectively. Constraints on the execution ordering of the children can be imposed; particularly partial orderings can be expressed elegantly. Additionally to this grammar, probabilities must be defined: for each plan the prior probability that it is executed, and given such a plan the conditional probability for certain observations.

At any time, there is a pending set of actions. Each action in this set is a candidate for the next observed action, because it would contribute to the agent's goal and it is "enabled" by the previously executed action. Starting with an initial pending set, this set changes with each performed action. This execution model is a Hidden Markov Model, because the observer neither knows the goals nor the pending sets (Geib and Goldman, 2009). The actual plan recognition is then done by sampling the agent's goals and plans.

The pending set approach allows to recognize interleaved plans and plans with common goals.

2.1.4 Plan Recognition as Planning

Ramirez and Geffner (2009) established a new class of plan recognition systems. Instead of starting with a plan library, they require a library of goals an agent might have. A goal G from the goal library is then considered to be an agent's goal if there is some optimal plan that leads to G and is compatible with the observations.

The plan is determined by means of a STRIPS (e.g., Russell and Norvig, 2003) domain theory that describes available actions and their effects. An off-the-shelf planner is then applied to this planning domain with a candidate goal picked from the goal library. Besides A* and other optimal planning algorithms, also heuristic planners can be used to improve the approach's scalability.

As most other plan recognition systems, Ramirez and Geffner (2009) assume to observe actions directly. However, it is probably easy to integrate propositional observations with the presented approach.

Ramirez and Geffner (2010) extended their system to probabilistic plan recognition. The described goal library needs to be enriched with prior probabilities $\Pr(G)$ for each goal G and with probabilities $\Pr(O \mid G)$ for each observation given a certain goal. Using Bayes' rule, the posterior probabilities $\Pr(G \mid O)$ for G given the observations can be determined.

2.2 Temporal and Spatial Modeling

This section is concerned with spatio-temporal modeling and reasoning. At first, some qualitative approaches are presented. The second half deals with numerical approaches.

2.2.1 Qualitative Modeling

Most of the literature about temporal planning is about a qualitative representation (Nau et al., 2004).

Allen's (1983) Interval Algebra is based on intervals consisting of a starting and an ending timestamp and thirteen simple binary relations over intervals. These include, for example, *equal*, *before* and *overlap*. The algebra allows indefiniteness of two intervals' relation. For example, by requiring two intervals to be *before* or *after* another, one states they neither overlap nor meet each other. Therefore, there are 2^{13} possible relations between two intervals.

As an answer to Interval Algebra's intractability, Vilain et al. (1986) propose a Point Algebra. By only considering points, the Point Algebra reduces the number of simple relations to three, namely $<$, $=$ and $>$. Due to disjunctions, each pair of time points is in one of seven different relations. For example, the relation $<=$ specifies that the time points are either related by $<$ or $=$. Similarly, $<=>$ essentially says nothing and $<>$ expresses that they are unequal.

Generally, qualitative temporal and spatial reasoning systems are very similar and each can often be applied in the respective field of the other.

A calculus similar to Interval and Point Algebra but dedicated to spatial reasoning is the Region Connected Calculus (Randell et al., 1992). Regions can be said to be *disconnected*, *partially overlapping* or *(non)tangential proper part*s of each other, for example. However, the applicability of the Region Connected Calculus in the present domain appears to be questionable. For one, borders intuitively seem not to be that important in traffic. To put it crudely, whether two cars are partially overlapping or tangential or non-tangential proper parts of each other, the important thing is simple: they crashed. For another, the calculus looks rather one-dimensional; the relations express information on a very abstract level and not in a two-dimensional Cartesian plane, for example. The prevalent state in car traffic is the *disconnected* relation, while all other relations could be combined to *crashed*.

The Qualitative Trajectory Calculus (Van de Weghe et al., 2006) is an approach to combine qualitative temporal and spatial reasoning at once. It attempts to diversify the *disconnected* relation and incorporate movement. They introduce binary relations $+$, 0, $-$ where $+$ and $-$ represent propositions such as “faster” and “slower,” respectively. Assuming continuous change, the Qualitative Trajectory Calculus deduces that, if first + holds and then − holds at a later point in time, then 0 must hold sometime in between. This calculus has been used to model a passing maneuver by Van de Weghe et al. (2005). They basically draw a reference line between the overtaking car a and the slower car b. The movement is then modeled in terms of the direction of each car with respect to the reference line and with respect to a perpendicular line point. Assuming that both cars are in the same lane driving behind each other, the first − and + in the four-tuple $(-, +, +, -)$ express that a moves towards b and b moves away from a. Analogously, the second + and − mean that a moves to the right of the reference line and b moves to the left of the reference line. While the calculus provides a way to model the relative position and movement, the provided information seems to be simplistic. It seems that, given only a sequence of four-tuples, one cannot reconstruct the actual events on the road, because the cars' positions on the road are not known. For example, $(0, 0, -, +)$ may either mean that a and b are next to each other and a is faster than b, or it may mean that both vehicles drive at the same speed on the right lane and a changes to the left lane. Since Van de Weghe et al. (2005) give no practical results, it is not clear how the Qualitative Trajectory Calculus should be used. Additionally, the fact that a's *and* b's behavior to each other is represented in each four-tuple is not intuitive. In fact, one usually does not say that "b moves towards a" when a is faster than b. The expression is rather used when b is oncoming traffic.

2.2.2 Quantitative Modeling

In the present domain, a quantitative representation of time appears to be more reasonable due to physical effects. Moreover, not only time is variable, but parameters of actions may be, too. Time again depends on these parameters. For example, the effects of an action that sets the velocity of a car to a certain value at a specific point in time depend on both, the velocity and the time.

Approaches like temporal constraint networks (Dechter et al., 1991) seem not to be able to handle such dependencies. Their objective is to reason about metric time intervals with unary and binary constraints. A binary constraint

$$\begin{gathered} a_1 \leq X_j - X_i \leq b_1 \vee \\ \vdots \\ a_n \leq X_j - X_i \leq b_n \end{gathered}$$

restricts the time distance between the events X_i and X_j to be in one of the intervals $[a_i, b_i]$, $1 \leq i \leq n$. The class of simple temporal constraint satisfaction problems (TCSPs) defined by Dechter et al. (1991) is restricted to problems where for each pair of events at most one interval constraint exists. Simple TCSPs are solvable in polynomial time (Dechter et al., 1991).

Time constraint networks are based on having constant lower and upper bounds of all points in time. In the present domain, however, these bounds strongly depend on the physical values such as velocity and/or acceleration. When these are not constant but variables, too, binary constraints will not suffice. And even if the physical values are all known in advance, it might be possible but cumbersome to derive the lower and upper bounds.

More powerful tools like linear or even nonlinear constraints appear to be a better choice to express the physical relationships. It is not required to compute a solution that is optimal with respect to some objective function, though, because no such function exists. The question is rather whether or not the constraint system is consistent and to determine any solution.

Linear programs are known to be solvable efficiently. Although the most widely spread algorithm, Simplex, has an exponential worst-case runtime, linear programs can be solved in polynomial time with, among others, Karmarkar's interior point algorithm (Zimmermann, 2005, pp. 163ff).

There are various forms of nonlinear programming. In quadratic programming, the constraints are still linear, but the objective function may be quadratic (Zimmermann, 2005, pp. 208ff). Using quadratic programming, one can solve linear least-squares problems. This capability could be used to bring in line a number of measurements and a linear model. Another branch of nonlinear programming deals with nonlinear constraints and linear objective functions. In convex optimization, interior point methods are used for a subclass called semidefinite programming (Boyd and Vandenberghe, 2004). Boyd and Vandenberghe (2004) state that convex optimization has a wide range of applications, but it is not clear whether or not it lends itself to simple physical equations.

2.3 Automotive Scene Identification

The idea to recognize the current driving situation[1] is not entirely new. This section shortly presents some of this work.

Meyer-Delius et al. (2009) model each situation pattern with a Hidden Markov Model. Given a library of Hidden Markov Models $\lambda_1, \dots, \lambda_n$, the situation i is considered to be present as long as λ_i recognizes the generated state sequence. They distinguish between passing maneuvers, aborted passing maneuvers and following.

Stiller et al. (2008) describe situations in terms of first-order logic formulas. Using these formulas and Markov Logic Networks, they do probabilistic inference. The system's focus appears to lie on rather qualitative questions such as whether or not some thing is a pedestrian.

[1]In this section, the term "situation" is not used in the sense of the situation calculus.

An approach that seems to be similar to Ramirez and Geffner (2009) is presented by Dagli et al. (2002). They start off with a set of predefined goals and motivations for each driver and do planning to determine the actors' possible plans. Actions represent continuous physical processes that are either of longitudinal or latitudinal kind. Examples are *accelerate*, *remain longitudinal* from the longitudinal group and *lane change left/right* from the lateral from. A longitudinal and a lateral action may be executed synchronously; their temporal relation can be specified in terms of Allen's (1983) Interval Algebra. Apparently, actions and particularly action sequences do not have any formal semantics. Completion of an action is said to change the "discretely represented situation model," but (Dagli et al., 2002) lacks a detailed explanation of this change. Consequently, while sketching a number of ideas, Dagli et al. (2002) miss on giving a formal definition of their system.

Nagel and Arens (2005) represent situations by states of a finite state machine. In the terminology of Nagel and Arens (2005), these automata are situation graphs. Situation graphs can be refined hierarchically and thus constitute a tree of situation graphs. The chronology is defined by the possible transitions. Each situation has an associated propositional logic formula. The problem of plan recognition is then to find an execution of this automaton in synchronisation with the observations such that each situation's formula holds. A transition triggers at the moment the current state's formula does not hold anymore due to changed observations. To break the tie when multiple transitions could be triggered, transitions are labelled with priorities and the highest prioritized transition whose destination state's formula holds is taken.

2.4 Summary

Existing work on plan recognition is rather general. Much effort is put into how to formulate and represent plans and the relationships of actions among each other. Most work does not discuss the characteristics of different domains, but stick to discrete, rather coarse-grained example domains like kitchen behavior with a focus on cooking recipes (Kautz and Allen, 1986), computer security (Geib and Goldman, 2009), the shopping world or story understanding (Charniak and Goldman, 1991). The only exception is the human behavior tracking by Bui et al. (2002) which includes the trajectory of the human being as a non-discrete feature.

Furthermore, all approaches assume the existence of an action sensor instead of supporting some kind of observations of the environment. Some (e.g., Geib and Goldman, 2009) view determining the actions as a separate problem called activity recognition. This splits the whole job into two parts similar to how parsing is divided into lexical and syntactic analysis. However, in the present domain actions have continuous effects. This makes it difficult to deduce the actions directly from the observations. Keeping activity and plan recognition together allows to give hints to the activity recognition by pruning the search space to those actions that are actually allowed.

This idea is similar to DTGolog proposed in (Boutilier et al., 2000a,b) which aims at marrying agent programming with planning. The planning search space is narrowed by the general structure dictated by the program. In contrast, if activity and plan recognition are separated, the information flow is a one-way street.

Except for (Goultiaeva and Lespérance, 2006) and Ramirez and Geffner (2009), all mentioned approaches lack a notion of change induced by actions as provided by the situation calculus and STRIPS.

As a consequence, the situation calculus and Golog appear to solve many issues addressed directly or indirectly by the discussed papers. The situation calculus is a logic for reasoning about actions and change. Golog, as a high-level programming language based on the situation calculus, includes a well-structured and powerful representation of programs, i.e. plan templates.

Considering the existing work for temporal and spatial reasoning, it appears that quantitative approaches are the first choice. However, for higher level plan recognition the quantitative frameworks might be useful.

The published work to plan recognition specific to the field of driver assistance systems and autonomous driving is generally vague when it comes to the modeling part and the semantics.

Chapter 3

Foundations

This chapter introduces the situation calculus and Golog, a high-level programming language based on the situation calculus.

3.1 Situation Calculus

The situation calculus is a sorted second-order language to reason about dynamic systems (Reiter, 2001, p. 47) with *actions* and *situations*. Predicates and functions whose values depend on the situation are called *fluents* (relational or functional, respectively).

McCarthy's frame problem is to formalize which fluents are not changed by which actions (McCarthy and Hayes, 1969, pp. 30f). A solution of the frame problem due to Reiter (1991) is to associate a *successor state axiom* with each fluent. Such a successor state axiom is composed of a positive and a negative effect axiom of the fluent. For each relational fluent $F(\vec{x})$, the positive effect axiom $\gamma_F^+(\vec{x}, a, s)$ is intended to hold iff the action a executed in s "switches $F(\vec{x})$ on." Analogously, the negative effect axiom is intended to hold iff the action a executed in s "switches $F(\vec{x})$ off." There should be no situation s in which $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ both hold. The successor state axiom defines whether or not $F(\vec{x})$ holds in $do(a, s)$:

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

The definitions for functional fluents are omitted but analogous. The predicate $Poss(a, s)$ is, like the effect axioms, provided by the modeler and holds iff action a is possible in situation s. The situation constant S_0 denotes the initial situation which is also specified by the modeler.

A *basic action theory* $\mathcal{D}$ consists of some foundational axioms Σ, the unique name axioms $\mathcal{D}_{una}$, the first-order axiomatization of the initial world $\mathcal{D}_{S_0}$, precondition axioms $\mathcal{D}_{ap}$ and successor state axioms $\mathcal{D}_{ss}$ (Reiter, 2001, p. 60):

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss}.$$

The foundational axioms ensure that, for example, $S_0 \neq do(a, s)$. The unique name axioms ensure that two different action names actually mean different actions, i.e. the interpretation cannot map two names to the same objects.

With regression (Reiter, 2001, p. 61ff), situation terms can be eliminated in a goal formula. In order to be regressable, a formula must not mention situation terms that are not rooted in S_0, quantify over situations, all action arguments in $Poss$ must be some action term, and the formula must not use $=$ or $\sqsubseteq$, which denotes the predecessor relation. The original formula W and the regressed formula $\mathcal{R}[W]$ are equivalent:

$$\mathcal{D} \models W \equiv \mathcal{R}[W].$$

The result of regression is a formula whose only situation term is S_0. The regressed formula can be decided with $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$. Regression is a syntactical operation (Fritz and McIlraith, 2009).

3.2 Golog

Golog (Levesque et al., 1997) is a programming language based on the situation calculus.

The primitives of this language are the actions of the situation calculus. These actions are called primitive actions. The execution of a program that consists of a single action a succeeds if $Poss(a, s)$ holds in the current situation s. A more complex program succeeds if there is a path through the program consisting of primitive actions $(a_1; \ldots; a_n)$ such that $Poss(a_i, do(a_{i-1}, do(a_{i-2}, \ldots do(a_1, S_0) \ldots)))$ holds for all i.

A special kind of action is the test action $\phi?$ where ϕ is a situation-suppressed formula. It succeeds iff ϕ holds in the current situation. A test action is not a situation calculus action but a Golog language feature. Hence, the "execution" of test actions is not reflected in situation terms.

Constructs known from imperative languages are the sequence operator, conditional statements, loops and procedures. Nondeterministic operators include $\pi x \,.\, \sigma(x)$ which picks an argument x for σ, $\sigma_1 \,|\, \sigma_2$ branches to either σ_1 or σ_2, and σ^* repeats σ zero or more times.

Golog programs are intended to be executed in some situation s and lead to a situation s'. Execution is controlled by the ternary macro $Do(\delta, s, s')$ which expands into a second-order situation calculus formula that holds iff executing the Golog program δ in situation s can lead to situation s' (Reiter, 2001, p. 111). The Do macro is defined as follows:

$$\begin{aligned}
Do(\alpha, s, s') &\overset{\text{def}}{=} Poss(\alpha[s], s) \wedge s' = do(\alpha[s], s) \\
Do(\phi?, s, s') &\overset{\text{def}}{=} \phi[s] \wedge s = s' \\
Do(\sigma_1; \sigma_2, s, s') &\overset{\text{def}}{=} \exists s'' . Do(\sigma_1, s, s'') \wedge Do(\sigma_2, s'', s') \\
Do(\sigma_1 \,|\, \sigma_2, s, s') &\overset{\text{def}}{=} Do(\sigma_1, s, s') \vee Do(\sigma_2, s, s') \\
Do(\pi v . \sigma, s, s') &\overset{\text{def}}{=} \exists x . Do(\sigma^v_x, s, s') \\
Do(\sigma^*, s, s') &\overset{\text{def}}{=} \forall P . \big((\forall s_1 . P(s_1, s_1)) \wedge \\
&\qquad\qquad (\forall s_1, s_2, s_3 . Do(\sigma, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3))\big) \supset \\
&\qquad P(s, s').
\end{aligned}$$

In fact, this only defines the core of the language; conditional statements (*if*) and loops (*while*) can be easily defined using the given constructs, namely tests (ϕ?), nondeterministic iteration (*) and nondeterministic branch ($|$). Furthermore, Golog supports procedures which are defined below.

The syntax $\phi[s]$ and $a[s]$ restores the situation variable s in ϕ and a, respectively. The idea of suppressing situations is that one does not want to talk about situations in Golog programs as every relational and functional fluent call should automatically refer to the current situation. For example, the situation-suppressed action *buy*(*cheapest*) becomes *buy*(*cheapest*(s)), because *cheapest* is a functional fluent.

The necessity of second-order logic stems from the definition of the nondeterministic iteration σ^* and from the definition of procedures. In the above macro rule for iteration, P is the smallest relation of situations s and s' such that s' is reachable from s by zero or more executions of σ. This reflexive transitive closure is not first-order-definable which is why second-order logic is needed (Reiter, 2001, p. 112).

Second-order logic is also needed in order to define procedure calls (Reiter, 2001, p. 115). Imagine a sequence of procedure definitions $P_1(\vec{v}_1)\ \sigma_1, \ldots, P_m(\vec{v}_m)\ \sigma_m$ where P_i is the name, $\vec{v}_i$ the argument vector and σ_i the body of the i-th procedure. The helper macro definition

$$Do(P(\vec{t}), s, s') \overset{\text{def}}{=} P(\vec{t}[s], s, s')$$

defines that call to procedure P with arguments $\vec{t}$ in s leads to s' iff $P(\vec{t}[s], s, s')$ holds. P is then defined in second-order logic as the smallest set of tuples $(\vec{v}, s_1, s_2)$ closed under the execution of P's body:

$$\begin{aligned}
&Do(\{P_1(\vec{v}_1)\ \sigma_1, \ldots, P_m(\vec{v}_m)\ \sigma_m\}\ \sigma, s, s') \overset{\text{def}}{=} \\
&\quad \forall P_1, \ldots, P_m . \big(\bigwedge_{i=1}^{m} \forall \vec{v}, s_1, s_2 . Do(\sigma_i, s_1, s_2) \supset P_i(\vec{v}, s_1, s_2)\big) \supset Do(\sigma, s, s').
\end{aligned}$$

This semantics is called evaluation semantics. The advantage of defining *Do* as macro and not as ordinary predicate is that Golog programs are macro-expanded and never occur as logical terms. This keeps the theory simpler, because interpretations do not have to assign values to Golog program constructs and situation-suppressed fluents. On the downside, quantification over Golog programs is not possible. Closely related with this, *Do* allows only to execute a complete program and not just a part of it. Incremental program executing comes in handy for, e.g., interleaved concurrency and plan recognition.

The transition semantics defined by Giacomo et al. (2000) resolves these issues. Transition semantics is based on the predicate $Trans$ as opposed to the evaluation semantics' *Do* macro. $Trans(\sigma, s, \delta, s')$ succeeds if there is a transition from situation s to s' with respect to the first action of the program σ where δ is the rest of the program σ after this transition step. If this action is a test action $\phi?$, such a transition exists for $s' = s$ if the logical formula ϕ holds in s. If, on the other hand, the next action of σ is a primitive action a, a transition exists for s' being the a-successor situation of s if a is executable in s, i.e. $Poss(a, s)$ holds. In both cases, δ is the remainder of σ after executing its next action. The appendix of (Giacomo et al., 2000) shows how Golog programs can be encoded as first-order terms which is needed to quantify over programs; the following definition of $Trans$ abstracts from this:

$$\begin{aligned}
Trans(Nil, s, \delta, s') &\equiv False \\
Trans(\alpha, s, \delta, s') &\equiv \delta = Nil \land Poss(\alpha[s], s) \land s' = do(\alpha[s], s) \\
Trans(\phi?, s, \delta, s') &\equiv \delta = Nil \land \phi[s] \land s' = s \\
Trans(\pi v \,.\, \sigma, s, \delta, s') &\equiv \exists x \,.\, Trans(\sigma^v_x, s, \delta, s') \\
Trans(\sigma_1; \sigma_2, s, \delta, s') &\equiv \exists \sigma'_1 \,.\, Trans(\sigma_1, s, \sigma'_1, s') \land \delta = [\sigma'_1, \sigma_2] \lor \\
&\qquad Final(\sigma_1, s) \land Trans(\sigma_2, s, \delta, s') \\
Trans(\sigma_1 \,|\, \sigma_2, s, \delta, s') &\equiv Trans(\sigma_1, s, \delta, s') \lor \\
&\qquad Trans(\sigma_2, s, \delta, s') \\
Trans(\sigma^*, s, \delta, s') &\equiv \exists \delta' \,.\, Trans(\sigma, s, \delta', s') \land \delta = [\delta', \sigma^*].
\end{aligned} \tag{3.1}$$

Additionally to $Trans$, a $Final$ predicate is needed to determine when program execution may stop:

$$\begin{aligned}
Final(Nil, s) &\equiv True \\
Final(a, s) &\equiv False \\
Final(\phi?, s) &\equiv False \\
Final(\pi v \,.\, \sigma, s) &\equiv \exists x \,.\, Final(\sigma^v_x, s) \\
Final(\sigma_1; \sigma_2, s) &\equiv Final(\sigma_1, s) \land Final(\sigma_2, s) \\
Final(\sigma_1 \,|\, \sigma_2, s) &\equiv Final(\sigma_1, s) \lor Final(\sigma_2, s) \\
Final(\sigma^*, s) &\equiv True.
\end{aligned}$$

In fact, the given definitions of $Trans$ and $Final$ are incomplete: Giacomo et al.

(2000) also define conditional statements (*if*) and loops (*while*) by means of *Trans* and *Final*. The reason is that by this, the respective branching condition and the first action of the branch can be executed atomically, that is, without any concurrently running program getting in between. This is called the synchronized if-then-else and while-loop. However, we refrain from these features in the following in order to keep the formulas shorter and clearer. Furthermore, Section 5.4.4 provides a similar language feature, which could be used to transform conditional statements and loops into their synchronized versions.

The behavior of the classic *Do* macro can be simulated using the reflexive transitive closure of *Trans* and asserting that the resulting rest program is final:

$$Do(\sigma, s, s') \overset{\text{def}}{=} Trans^*(\sigma, s, \delta, s') \land Final(\delta, s').$$

$Trans^*$ denotes the reflexive transitive closure of *Trans*:

$$\begin{aligned} Trans^*(\sigma, s, \delta, s') \overset{\text{def}}{=} \forall P \,.\, \big(& (\forall \sigma_1, s_1 \,.\, P(\sigma_1, s_1, \sigma_1, s_1)) \land \\ & (\forall \sigma_1, s_1, \sigma_2, s_2, \sigma_3, s_3 \,. \\ & \quad P(\sigma_1, s_1, \sigma_2, s_2) \land Trans(\sigma_2, s_2, \sigma_3, s_3) \supset \\ & \quad\quad P(\sigma_1, s_1, \sigma_3, s_3)\big) \supset \\ & P(\sigma, s, \delta, s'). \end{aligned}$$

The definitions of *Trans* and *Final* are bundled in $\mathcal{C}$. The theory is no longer just $\mathcal{D}$ but $\mathcal{D} \cup \mathcal{C}$, so a formula ϕ that mentions the new *Do* (and/or *Trans* etc.) holds iff $\mathcal{D} \cup \mathcal{C} \models \phi$.

3.3 Notation

Usually, uppercase roman letters denote predicates and constants, use of small letters represents variables and non-nullary functions. Variables without a quantifier are implicitly universally quantified. a often stands for a primitive action and s for a situation.

The Greek symbols σ, δ are used for possibly complex Golog programs. Variables of sort time (which will be introduced later) are represented by τ and probabilities by p. σ^v_x denotes the term σ with all occurrences of v substituted with x.

At a later point, we will distinguish between time-stamped and non-time-stamped primitive actions. The former is usually denoted by a, the latter by α. This matter will be amplified when time is introduced. Stochastic actions, which will be introduced later, too, are denoted by β. Finally, γ stands for an action that is either non-time-stamped primitive, stochastic or a test.

Golog programs will not mention situation terms. To restore the suppressed situation arguments in a test action $\phi?$, we write $\phi[s]$ for a situation s. Similarly, we write $\alpha[s, \tau]$ to restore s in all fluents mentioned in the non-time-stamped primitive action α, evaluate them at time τ and append a new timestamp parameter τ to α, thus making $\alpha[s, \tau]$ a time-stamped primitive action.

The notations $do(\vec{a}, s)$ and $do([a_1, \ldots, a_n], s)$ are shorthands for $do(a_n, \ldots do(a_1, S_0) \ldots)$.

The sequence operator associates to the right, that is, $\sigma_1; \sigma_2; \sigma_3 = (\sigma_1; (\sigma_2; (\sigma_3)))$.

The used logical operators are universal and existential quantification ($\forall$, $\exists$), conjunction and disjunction ($\wedge$, $\vee$), implication ($\supset$) and negation ($\neg$). The logical conjunction ($\wedge$) has higher binding precedence than disjunction, e.g. $\phi \wedge \psi \vee \chi \overset{\text{def}}{=} (\phi \wedge \psi) \vee \chi$. Bracket usage is reduced with the dot-notation unless it is confusing, e.g. $\phi \wedge \exists x \,.\, \psi \overset{\text{def}}{=} \phi \wedge ((\exists x)\psi)$.

Chapter 4

Modeling

The goal of this chapter is to explore modeling in the car traffic domain from the perspective of plan recognition. Basic action theories for the rudimentary physics of a vehicle are developed. It is exemplarily shown how a plan library could be constituted using such a basic action theory's primitive actions and fluents.

The following example models a passing maneuver on a two-lane road. This scenario is well-suited as example because it requires time and continuous fluents. Additionally, it involves multiple drivers and is safety-critical.

Compared to the real world, the modeling world is simplified in that actions like changing acceleration, velocity or yaw are executed instantaneously. Figure 4.1 shows a car (solid) overtaking another vehicle (hatched). The dashed and dotted rectangles represent an instantaneous change of the overtaking car's yaw.

At some points in the following, actions will have a parameter τ. This parameter represents the time at which an action is executed. Generally, the idea is not to specify this time at the modeling level. At execution time, these parameters are added. As a consequence, the actions appearing in successor state axioms and precondition axioms are time-parameterized. The exact definition of time is given in Section 5.1.

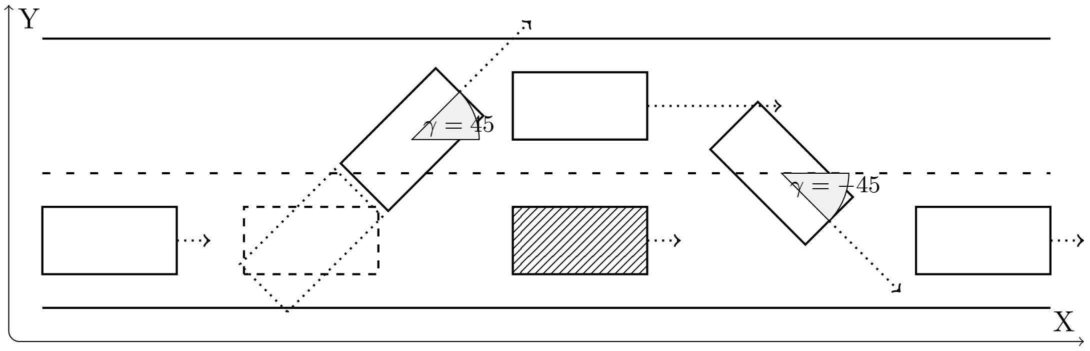


Figure 4.1: Passing maneuver with instantaneous changes of velocity and yaw.

4.1 Passing Maneuver with Instantaneous Velocity and Yaw

The first attempt to model a passing maneuver makes use of the two actions

- $setYaw(\gamma)$ with angle $\gamma \in [-\pi, \pi)$ and
- $setVeloc(v)$ with velocity $v \in \mathbb{Q}$

to instantaneously change the orientation of the car with respect to the road and to change the velocity, respectively. For both of these actions, there is a functional fluent, *yaw* and *veloc*, respectively, which is only changed by the respective action.

The third functional fluent, *pos*, denotes the position of a given car. The position of a car is represented in simplified terms as point in a two-dimensional Cartesian coordinate system. As depicted in Figure 4.1, the Y-coordinate is the lateral position, and the X-coordinate is the longitudinal position of the car.

In contrast to *veloc*, *pos* is a continuous fluent. This means that the return type of *pos* is a function of time like in cc-Golog (Grosskreutz and Lakemeyer, 2003). While the definition of this function is constant per situation, it can be evaluated with different points in time during a single situation. *pos* thus yields different positions at different specific points in time in a single situation.

These functions are restricted to be at most polynomial for practical reasons. In the presented example, they are even linear. In logic, polynomial functions can be represented as terms with functors *constant*, *linear* etc. In general, a polynomial $\sum_{i=0}^{n} a_i \cdot (\tau - \tau_0)^i$ would be represented as term $poly^n(a_0, \ldots, a_n, \tau_0)$. Variables that denote such a term have a superscript t, e.g. e^t. A term e^t can be evaluated at a time τ with the function $val(e^t, \tau)$.

Since *veloc* should simply return the velocity set by the last *setVeloc* action, its successor state axiom is:

$$\begin{aligned} v = veloc(do(a, s)) \equiv {} & (\exists \tau) a = setVeloc(v, \tau) \lor \\ & v = veloc(s) \land (\forall v', \tau) a \neq setVeloc(v', \tau). \end{aligned}$$

Time does not play any role in the behavior of *veloc*. The definition of *yaw* is analogous.

The fluent *pos* returns a tuple of functions of time for the X- and Y-coordinates of the car. Evaluated at a point in time, these functions return the two-dimensional position of the car at this time. If the velocity or yaw of the car is changed (instantaneously) at time τ_0, the functions change. Then, for example, the X-coordinate is the sum of the old X-coordinate at time τ_0 $(*)$ and the change of the X-coordinate due to the (possibly new) velocity and the (possibly new) yaw ($**$). This can be expressed semi-formally by

$$x(\tau) = \underbrace{x(\tau_0)}_{(*)} + \underbrace{\cos(yaw(\tau_0)) \cdot veloc(\tau_0) \cdot (\tau - \tau_0)}_{(**)} .$$

The function of the Y-coordinate is analogous. In the following successor state axiom, $start(s)$ denotes the time at which the situation s starts, or, in other words, the time at which the change of velocity or yaw occurs. The actual definition of $start$ is not relevant for now. Hence, the successor state axiom for pos is:

$$\begin{aligned}(x_1^t, y_1^t) = pos(do(a, s)) \equiv \exists \tau_0, x_0^t, y_0^t, x_0, y_0, v, \gamma \,.\quad & \\ & \big(a = setVeloc(v, \tau_0) \land \gamma = yaw(s) \lor \\ & \quad a = setYaw(\gamma, \tau_0) \land v = veloc(s)\big) \land \\ & (x_0^t, y_0^t) = pos(s) \land \\ & x_0 = val(x_0^t, \tau_0) \land \\ & y_0 = val(y_0^t, \tau_0) \land \\ & x_1^t = linear(x_0, \cos(\gamma) \cdot v, \tau_0) \land \\ & y_1^t = linear(y_0, \sin(\gamma) \cdot v, \tau_0) \lor \\ & (x_1^t, y_1^t) = pos(s) \land \\ & (\forall v, \tau) a \neq setVeloc(v, \tau) \land \\ & (\forall \gamma, \tau) a \neq setYaw(\gamma, \tau).\end{aligned}$$

In Figure 4.2, the procedure *overtake* models a passing maneuver in the scenario sketched in this section. Actions and fluents have an additional parameter that represents the acting driver. *onLeft/RightLane* and *behind* can be defined as macros using the *pos* fluent, e.g.

$$onLeftLane(v, s, \tau) \overset{\text{def}}{=} \exists x^t, y^t \,.(x^t, y^t) = pos(v, s) \land 0\,\text{m} < val(y^t, \tau) \leq 5\,\text{m}.$$

To improve readability, the syntax is assimilated to classical programming languages a bit, e.g., **loop** and **pick** stand for σ^* and $\pi v \,.\, \sigma$, respectively. There are two threads of execution which are delimited by the keyword **concurrently with**. The first thread cares about changing the lane, the second manages the speed. This program is highly nondeterministic:

- nondeterministic concurrency ($\sigma_1 \parallel \sigma_2$, cf. Section 5.2),
- nondeterministic choice of an argument for each primitive action ($\pi v \,.\, \sigma$), and
- nondeterministic iteration of each primitive action (σ^*).

The idea of the latter two kinds of nondeterminism is that drivers are expected to dangle a bit during the lane change. Section 5.3 argues that nondeterminism should not lead to counter-intuitive behavior like in cc-Golog (Grosskreutz and Lakemeyer,

```
proc $overtake(V, W)$
    $behind(V, W)$ ?;
    $onRightLane(V)$ ?;
    $onRightLane(W)$ ?;
    begin
        loop                                    % steer to the left lane
            pick $\gamma$ do
                $\gamma \geq 0°$ ?;
                $setYaw(V, \gamma)$
            endpick
        endloop;
        $setYaw(V, 0°)$;
        $onLeftLane(V)$ ?;
        wait for $behind(W, V)$;
        loop                                    % steer back to the right lane
            pick $\gamma$ do
                $\gamma \leq 0°$ ?;
                $setYaw(V, \gamma)$
            endpick
        endloop;
        $setYaw(V, 0°)$;
        $onRightLane(V)$ ?
    concurrently with
        loop                                    % in the meantime, accelerate
            pick $v$ do
                $setVeloc(V, v)$
            endpick
        endloop
    end;
    $onRightLane(W)$ ?;
    $behind(W, V)$ ?
endproc
```

Figure 4.2: Passing maneuver of V against W.

2003). Nondeterminism blows up the search space, of course, which is why Section 5.4 introduces other ways of achieving robustness.

4.2 Improving Realism with Preconditions

The instantaneous actions used in the previous section are generally unrealistic. However, using preconditions, it is possible to enforce some plausibility, while still keeping the equations that model the physics linear. As explained in Section 2.2.2, linear (in)equations have the advantage over nonlinear constraints that they can be solved efficiently.

For example, if the car is stopped at the moment τ_0 and a $setVeloc(30\,\text{m/s})$ occurs at time τ, it is clear that $\tau - \tau_0 \geq 5\,\text{s}$, because the top acceleration of fast cars is about $6\,\text{m/s}^2$. Similarly, the deceleration in cars is at most $10\,\text{m/s}^2$. The following precondition results:

$$Poss(setVeloc(v,\tau),s) \equiv \tau \neq start(s) \wedge$$
$$-10\,\text{m/s}^2 \leq \frac{v - val(veloc(s), start(s))}{\tau - start(s)} \leq 6\,\text{m/s}^2.$$

Due to the monotonicity of time, which will be axiomatized in Section 5.1, $\tau \geq start(s)$ holds. Hence, the inequations can be expressed as sums

$$-v + val(veloc(s), start(s)) - 10\,\text{m/s}^2 \cdot \tau + 10\,\text{m/s}^2 \cdot start(s) \leq 0\,\text{m/s} \text{ and}$$
$$v - val(veloc(s), start(s)) - 6\,\text{m/s}^2 \cdot \tau + 6\,\text{m/s}^2 \cdot start(s) \leq 0\,\text{m/s}.$$

Obviously, these inequation is linear.

A similar heuristic can be used for $setYaw$. In order to achieve a yaw γ, a car can drive along the segment of a circle d with some radius r as visualized in Figure 4.3. The driver has to drive along the thick circular segment d until he reaches the tangent point. The length of this segment is $d = \alpha \cdot r$. Since $\delta = 90° - \gamma$, $\beta = \delta$ and $\alpha = 90° - \beta = \gamma$, for a given speed v it takes time $\frac{\gamma \cdot r}{v}$ to achieve a yaw of γ. Generally, radius r of this circle grows with the car's speed v. An approximation is $r(v) = r_0 + \frac{v^2}{g \cdot \mu}$ where $r_0 = 5\,\text{m}$ is the turning radius of the car, $g = 9.81\,\text{m/s}^2$ is the gravitational acceleration and $\mu = 0.8$ is the stiction (all numbers are just estimations). For velocity v and yaw γ, $t(v,\gamma) = \frac{\gamma \cdot r(v)}{v}$ is the minimum time needed to achieve yaw γ. Figure 4.4a displays $t(v,\gamma)$. The resulting precondition axiom is simple:

$$Poss(setYaw(\gamma,\tau),s) \equiv \tau - start(s) \geq t(val(veloc(s), start(s)), \gamma).$$

Unfortunately, this inequation is only linear if at least $veloc$ or γ is constant.

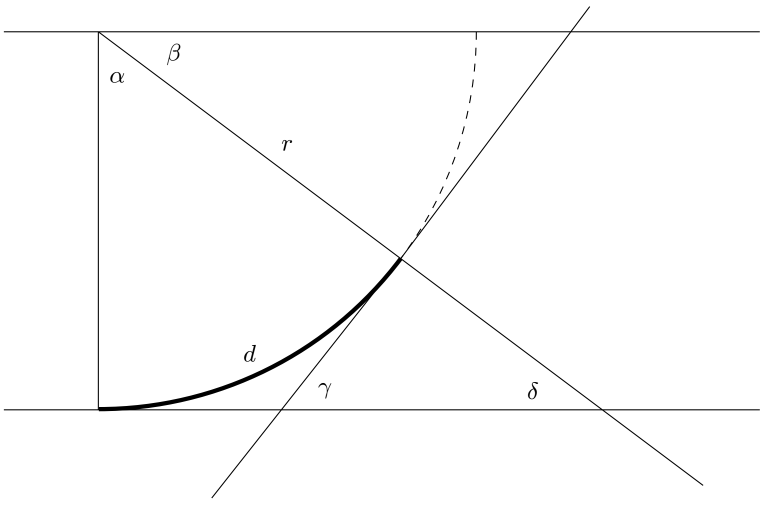


Figure 4.3: Trajectory d to achieve yaw γ.

To keep things linear and the solution space convex, the constraint $\tau \geq t(v, \gamma)$ could be approximated with planes in the intervals $v \in [7\,\text{m/s}, 60\,\text{m/s}]$ (which is about 25 km/h to 216 km/h) and for $\gamma \in [0, \pi]$ (γ can be 180° if the driver wants to change from −90° to 90°). The approximation shown in Figure 4.4b consists of the two tangential planes in points $(7, 0)$ and $(60, \pi)$

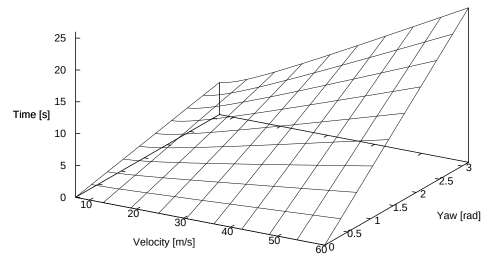
$$\tau \geq t(7,0) + (v-7) \cdot \frac{\partial t(v,\gamma)}{\partial v}(7,0) + (\gamma - 0) \cdot \frac{\partial t(v,\gamma)}{\partial \gamma}(7,0)$$
$$\tau \geq t(60,\pi) + (v-60) \cdot \frac{\partial t(v,\gamma)}{\partial v}(60,\pi) + (\gamma - \pi) \cdot \frac{\partial t(v,\gamma)}{\partial \gamma}(60,\pi).$$

Multiple constraints $\tau \geq c_1 \cdot v + c_2 \cdot \gamma + c_3$ have the effect of taking the maximum of all approximating planes.

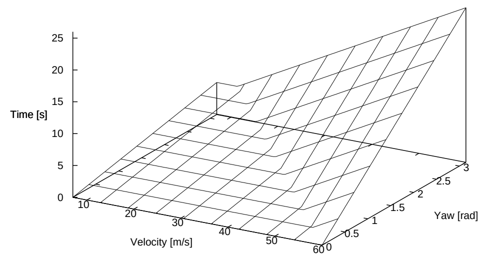
4.3 Passing Maneuver with Instantaneous Acceleration and Yaw

Actions do not have to be as high-level as *setVeloc*. For example, acceleration could be changed instantaneously with an action *setAccel*.

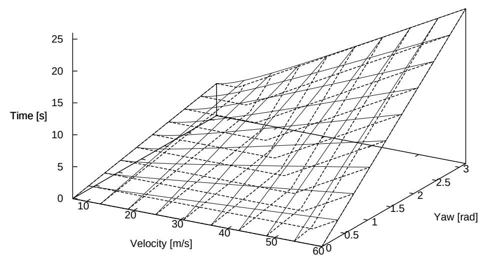
Then, *veloc* simply returns the value set by the last *setVeloc* but becomes a continuous



(a) Exact function $t(v, \tau)$.



(b) Approximations with maximum of two tangential planes.



(c) Exact and approximated (dashed) function.

Figure 4.4: Dependency of time on velocity and yaw.

fluent. The successor state axiom is:

$$\begin{aligned} v_1^t = veloc(do(a, s)) \equiv \exists \tau_0, f, v_0^t, v_0 \,. \\ & a = setAccel(f, \tau_0) \land \\ & v_0^t = veloc(s) \land \\ & v_0 = val(v_0^t, \tau_0) \land \\ & v_1^t = linear(v_0, f, \tau_0) \lor \\ v_1^t = veloc(s) \land (\forall f, \tau) a \neq setAccel(f, \tau). \end{aligned}$$

The situation gets worse when looking at *pos*: the degrees of the polynomials in its successor state axiom increase by one, too, and hence become quadratic functions:

$$\begin{aligned} (x_1^t, y_1^t) = pos(do(a, s)) \equiv \exists \tau_0, x_0^t, y_0^t, x_0, y_0, v, f, \gamma \,. \\ & \big(a = setAccel(f, \tau_0) \land \gamma = yaw(s) \lor \\ & \quad a = setYaw(\gamma, \tau_0) \land f = accel(s)\big) \land \\ & (x_0^t, y_0^t) = pos(s) \land \\ & x_0 = val(x_0^t, \tau_0) \land \\ & y_0 = val(y_0^t, \tau_0) \land \\ & v = val(veloc(s), \tau_0) \land \end{aligned}$$

$$
\begin{aligned}
&\quad x_1^t = \mathit{quadratic}(x_0, \cos(\gamma) \cdot v, \tfrac{1}{2} \cdot \cos(\gamma) \cdot f, \tau_0) \wedge \\
&\quad y_1^t = \mathit{quadratic}(y_0, \sin(\gamma) \cdot v, \tfrac{1}{2} \cdot \sin(\gamma) \cdot f, \tau_0) \vee \\
&(x_1^t, y_1^t) = \mathit{pos}(s) \wedge \\
&\quad (\forall f, \tau) a \neq \mathit{setAccel}(f, \tau) \wedge \\
&\quad (\forall \gamma, \tau) a \neq \mathit{setYaw}(\gamma, \tau).
\end{aligned}
$$

Nonlinear functions raise problems when it comes to solving systems of inequations. Hence, caution should be exercised when introducing polynomial functions. For this very reason, we stick with linear constraints in our implementation and examples.

4.4 Summary

This chapter demonstrated how a program in a plan library for car traffic might look like. The two main components of such a plan library are

- the basic action theory consisting of primitive actions and fluents with their respective preconditions and successor state axioms, and
- programs using these actions and fluents.

Finding a good basic action theory is not trivial and depends on the application. The general tradeoff between expressiveness and tractability of arithmetic constraints complicates this task.

This thesis focuses on the formal definition of a Golog dialect as modeling language for plan recognition. The language as such imposes no limitations on the types of constraints.

For practical purposes, however, one might want to stick with linear constraints as far as possible. Section 4.2 has shown that some of the characteristics of car traffic can be approximated using linear inequations.

Chapter 5

Semantics

This thesis' approach to plan recognition is, given a sequence of observations and a candidate program, to search for an execution of the program which entails all observations. In a sense, the real world happenings represented by the observations are (tried to be) replicated in the fictional model. If the fictional execution is congruous with the observations, the program is assumed to be executed in reality and the plan is recognized. In its final stage, the system not only returns definitive yes/no answers, but assigns confidences to the candidate programs instead.

This chapter is about the language in which such candidate programs are written. For one, the language needs to support modeling as done in Chapter 4. For another, the execution model must somehow integrate the execution of programs with checking whether or not observations holds.

The proposed language is a dialect of Golog (cf. Section 3.2). A central issue is the handling of time. The goal is to pair descriptive modeling in programs with quantitative timestamps in situation terms.

Further topics of this chapter are concurrency and nondeterminism in Section 5.2 and Section 5.3, respectively. We argue that in this Golog dialect, both features can coexist without leading to anomalies.

Section 5.4 shows that robustness can be modeled using probabilities and decision theory. This gives rise to the aforementioned confidences that a certain candidate program explains the observations. Also, this semantics allows to define atomic complex actions, that is, sub-programs that are executed isolated from concurrently running programs.

5.1 Time

Chapter 4 sketched when and how time could be used. In fact, the presented program did not contain any reference to time at all. However, the successor state axiom for *pos* used time to determine the current position, and preconditions even imposed constraints on the timing. The definition of time, namely the function *start*, was left unclear in the previous chapter. This section discusses time in the situation calculus and proposes a notion of time that allows modeling as done in the previous section.

5.1.1 Temporal Sequential Golog and cc-Golog

There are different extensions of the situation calculus and Golog in order to support time. In temporal sequential Golog (Reiter, 1998), each action is parameterized with the time at which it is executed. As a consequence, time is modeled rather explicitly in the basic action theories and situation terms contain timestamps.

In contrast to this notion of time, cc-Golog (Grosskreutz and Lakemeyer, 2000a) provides a special primitive action, *waitFor*, which controls the lapse of time. $waitFor(\phi)$ advances to the earliest upcoming point in time at which ϕ holds.

With cc-Golog, a program to overtake another car v on a highway could end with an action sequence like

$$\ldots; waitFor(distanceTo(v) \geq securityDistance); changeLaneRight.$$

Due to *waitFor*'s behavior in cc-Golog, the overtaking car would execute *changeLaneRight* directly after it has reached the security distance. This is called least time point semantics and means that each *waitFor* action is executed as early as possible. In reality, however, drivers usually wait a little longer before they go back into the lane. Hence, cc-Golog's semantics of *waitFor* is not the right choice for plan recognition.

Temporal sequential situation terms match observations pretty well, because both have an explicit associated timestamp. But cc-Golog's implicit and descriptive model of time appears to be more natural for the modeling part in a continuous environment such as automotive domains.

5.1.2 Combining Temporal Sequential Golog and cc-Golog

The presented model of time is intended to bridge both approaches: in our Golog dialect, time is constrained in programs descriptively with *waitFor* (and/or similar) actions in the fashion of cc-Golog. At execution side, time is allowed to pass indefinitely after each action. In fact, time advances at least to the point at which the next primitive action is executable. The time at which a primitive action is executed is encoded as parameter of this action in the same way like in temporal sequential Golog.

To implement the described model, the rule for primitive actions in the transition semantics is replaced with the following definition:

$$\begin{aligned} Trans(\alpha, s, \delta, s') \equiv {} & \delta = Nil \wedge \\ & \exists \tau \,.\, \tau \geq start(s) \wedge \\ & Poss(\alpha[s, \tau], s) \wedge \\ & s' = do(\alpha[s, \tau], s). \end{aligned} \tag{5.1}$$

Recall that α denotes a single, non-time-stamped primitive action. The notation $\alpha[s, \tau]$ restores s and τ in all fluents that occur in α, and also appends the timestamp τ as additional parameter to the action α. The given $Trans$ axiom imposes a monotonicity constraint on time. Additionally, the precondition axiom of $\alpha[s, \tau]$ may constrain τ further. As long as τ meets the monotonicity constraint ($\tau \geq start(s)$) and potential constraints of $Poss$, time may pass freely. The function $start$, which denotes the starting time of a situation, is defined as

$$start(S_0) = \mathrm{T}_0 \text{ and } start(do(a, s)) = time(a)$$

where T_0 is a constant such as 0 defined in $\mathcal{D}_{S_0}$ and $time$ extracts the timestamp of an action, i.e. $time(\alpha[s, \tau]) = \tau$. The $time$ function is borrowed from sequential temporal Golog (Reiter, 2001, p. 152ff). For each action $A(\vec{x}, \tau)$, an axiom $time(A(\vec{x}, \tau)) = \tau$ must be added to the basic action theory $\mathcal{D}$. Furthermore, the definition of $start$ is added to $\mathcal{D}$.

In cc-Golog, only the $waitFor$ action induces lapse of time and this is realized in cc-Golog's $start$ function. In contrast, with the described semantics, any action can control time using its precondition axiom, because time goes by freely modulo the potential constraints imposed by $Poss$. For example, $waitFor$ could be defined with the precondition

$$Poss(waitFor(\phi, \tau), s) \equiv \phi[s, \tau].$$

The notation $\phi[s, \tau]$ stands for the logical formula ϕ with all suppressed situations restored by s and all occurrences of continuous fluents evaluated with val, e.g. $(f(\vec{x}) < 3)[s, \tau] = (val(f(\vec{x}, s), \tau) < 3)$. Like every action, $waitFor$ is given an additional time parameter by the above $Trans$ axiom. cc-Golog's $waitFor$ does not have a time parameter; instead, $start$ unwinds the situation term back to the last $waitFor$ action and checks for which time in point the condition holds.

The new sort of timestamps may be, for example, the real or rational numbers.

In cc-Golog, $waitFor$ actions have least time point semantics (Grosskreutz and Lakemeyer, 2000a). This means that $waitFor(\phi)$ is executed at the earliest point in time τ at which $\phi[s, \tau]$ holds. This can be achieved with the above semantics by modifying the $Poss(waitFor(\phi, \tau), s)$ formula given above to

$$Poss(waitFor(\phi, \tau), s) \equiv \phi[s, \tau] \wedge (\forall \tau')(start(s) \leq \tau' \wedge \tau' < \tau \supset \neg\phi[s, \tau']).$$

That precondition fails same like cc-Golog's when no least time point exists like in conditions such as $\tau > 3$. However, probably the benefit of the least time point semantics would be rather small, because all subsequent non-$waitFor$ actions are neither executed immediately nor do they underlie the least time point semantics unless that is explicitly expressed in their preconditions. So time could still pass indefinitely after a $waitFor$ action. On the downside, the least time point semantics leads to anomalies with concurrent nondeterministic programs (Grosskreutz and Lakemeyer, 2003). Dropping the least time point semantics allows for nondeterministic programs as explained in Section 5.3.

The proposed changes to the semantics do not make test actions obsolete. While $waitFor$ actions are primitive actions and therefore occur with an associated timestamp in situation terms, test actions are "timeless" and never occur in situation terms. A test may quite mention continuous fluents, but they are not evaluated automatically. Instead, a continuous fluent returns a term such as $linear(1, 2, 3)$, which may then be evaluated using val at a specific point in time. In contrast to tests, $waitFor$ actions implicitly evaluate continuous fluents by plugging in the action's execution timestamp.

In contrast to cc-Golog, however, $waitFor$ is not a special primitive action anymore but can be easily defined by the user. The reason is that control of time is moved from cc-Golog's $start$ function to the actions' preconditions (and $Trans$ additionally ensures monotonicity of time).

This notion of time can be summarized as follows:

- Actions in programs are timeless.
- When executed, actions are passed to $Trans$.
- $Trans$ adds a time parameter to each primitive action which is constrained by
  - $Trans$ itself to ensure that time increases monotonically, and
  - the action's precondition like in $waitFor$ or as shown in Section 4.2.
- The time-stamped primitive actions form situation terms.

5.2 Concurrency

As mentioned in the introduction, different actors typically act simultaneously on the road. This kind of concurrency is not needed by the modeler, because the programs in the plan library refer to a isolated drivers only. But the language still has to support concurrency in order to reflect that plans are executed concurrently in the situation terms.

If n actors act simultaneously, the goal of plan recognition is to find a program $\sigma = \sigma_1 \parallel \ldots \parallel \sigma_n$ which explains the observations, that is, find an execution of the concurrent programs such that all observations are entailed. In which interleaved order the σ_i are executed, is not known except for the constraints imposed by the observations. Which interleaving of the σ_i actually appears in reality is not even known, so the semantics of $\parallel$ can be defined to be any interleaving of its operands:

$$\begin{aligned} \mathit{Trans}(\sigma_1 \parallel \sigma_2, s, \delta, s') \equiv\ & \exists \delta' \,.\, \mathit{Trans}(\sigma_1, s, \delta', s') \wedge \delta = \delta' \parallel \sigma_2 \vee \\ & \exists \delta' \,.\, \mathit{Trans}(\sigma_2, s, \delta', s') \wedge \delta = \sigma_1 \parallel \delta' \\ \mathit{Final}(\sigma_1 \parallel \sigma_2) \equiv\ & \mathit{Final}(\sigma_1) \wedge \mathit{Final}(\sigma_2). \end{aligned} \tag{5.2}$$

This is the same like the semantics of $\parallel$ in ConGolog (Giacomo et al., 2000).

This concurrency operator can also be used to model that a single driver does multiple things concurrently, of course. This is the second kind of concurrency mentioned in Section 1.2. It is actually used in Figure 4.2 on page 26 to express that accelerating happens sometime during the lane changes. If an actor performs multiple things in parallel, such as indicating during a lane change, this can be very well modeled with processes. Processes consist of a pair of actions *startAction* and an *endAction* and a fluent *acting* which is true between the start and end of the process.

5.3 Nondeterminism

In cc-Golog, nondeterministic features of Golog are disallowed due to their counter-intuitive behavior with respect to cc-Golog's semantics (Grosskreutz and Lakemeyer, 2003, pp. 193, 196f). The reason is that in cc-Golog, $\sigma_1 \parallel \sigma_2$ executes a first action of σ_1 or σ_2 depending on which one is possible earlier:

$$\begin{aligned} \mathit{Trans}(\sigma_1 \parallel \sigma_2, s, \delta, s') \equiv \neg \mathit{Final}(\sigma_1, s) \wedge \neg \mathit{Final}(\sigma_2, s) \wedge \big(& \\ \exists \delta_1 \,.\, \mathit{Trans}(\sigma_1, s, \delta_1, s') \wedge \delta = \delta_1 \parallel \sigma_2 \wedge & \\ \big(\forall \delta_2, s_2 \,.\, \mathit{Trans}(\sigma_2, s, \delta_2, s_2) \supset \mathit{start}(s') \leq \mathit{start}(s_2)\big) \vee & \\ \exists \delta_2 \,.\, \mathit{Trans}(\sigma_2, s, \delta_2, s') \wedge \delta = \sigma_1 \parallel \delta_2 \wedge & \\ \big(\forall \delta_1, s_1 \,.\, \mathit{Trans}(\sigma_1, s, \delta_1, s_1) \supset \mathit{start}(s') < \mathit{start}(s_1)\big)\big). & \end{aligned}$$

Hence, interleaving of actions itself is deterministic. This semantics is intended in scenarios like the following (Grosskreutz and Lakemeyer, 2003, p. 193):

$$\underbrace{(\mathit{waitFor}(\mathit{batteryIsEmpty}); \mathit{charge})}_{\sigma_1} \parallel \underbrace{\mathit{doSomething}}_{\sigma_2}.$$

The left program, σ_1, should only be executed when the battery is empty. Waiting until the battery is empty doing nothing and then going to charge is obviously not the intended behavior.

Nondeterministic features like the branch | do not work with this semantics: Assume that primitive action a is possible at time 1, b at 2 and c at 3. Let the program be $(a \mid c) \parallel b$. One has to consider both parts of the disjunction in cc-Golog's concurrency semantics to see the anomaly. When the interpreter first executes $a \mid c$ and there branches to a, it has to assert that all situations resulting from b cannot occur earlier than $do(a, S_0)$. Since

$$1 = start(do(a, S_0)) \leq start(do(b, S_0)) = 2$$

holds, this is the case and hence, $do([a, b], S_0)$ is reachable. But if b is executed first, the interpreter has to check that all situations that result from $a \mid c$ occur later than $do(b, S_0)$. This is not the case, because $do(a, S_0)$ is reachable by $a \mid c$ and

$$2 = start(do(b, S_0)) < start(do(a, S_0)) = 1$$

fails. Therefore, $do([b, c], S_0)$ is not entailed by cc-Golog's semantics of concurrency, even though it conforms to the monotonicity of time $start(do(b, S_0)) \leq start(do([b, c], S_0))$. For this reason, nondeterministic features are not part of cc-Golog.

Since the semantics described in Section 5.1 does not include cc-Golog's least time point concept, ConGolog's definition of interleaved concurrency as defined in Section 5.2 suffices. Nondeterministic language constructs like $\parallel$ and π seem not to lead to unintuitive results. Due to existentially quantified time, the interpreter has the freedom to search for any execution (modulo the program structure) that is feasible.

In reality, it might be better to refrain from nondeterminism for performance reasons. This section simply argued that nondeterminism does not interfere with the other language features.

5.4 Robustness

Section 1.2 distinguished three relevant kinds of robustness: robust timing constraints, optionality of some types of actions, and tolerance towards deviations of observation and expected fluent values.

Timing robustness is an inherent part of the notion of continuous time presented in Section 5.1.

Optional actions can be achieved with simple nondeterminism like

$$\ldots; (optionalAction \mid Nil); \ldots.$$

To save space, a keyword *opt* could be introduced with the following semantics:

$$\begin{aligned} Trans(opt(\sigma), s, \delta, s') &\equiv Trans(\sigma \mid Nil, s, \delta, s') \\ Final(opt(\sigma)) &\equiv True. \end{aligned}$$

An example for the use of *opt* is the "optional" indicating during a lane change.

This section deals with fluent value robustness. The goal is to prepare the language to handle discrepancies between reality, that is observations, on the one side and the model's fluent values on the other side. How this semantics pays off is shown in Section 6.5.

It might be desirable to measure the deviation somehow quantitatively in order to obtain a confidence in a certain plan explaining the observations. This would of course imply abandoning the pure consistency-basedness of the approach.

In the following, existing work related to robustness is discussed. Then, an approach is developed based on some of this work.

5.4.1 Non-Probabilistic Robustness

The *overtake* procedure from Figure 4.2 on page 26 tackles the problem using excessive nondeterminism: the lane change of a car is explained by a nondeterministic repetitions of nondeterministically picking an angle $\gamma \geq 0$ and performing $setYaw(\gamma)$. Consequently, for any sequence of observations of a lane changing car a corresponding sequence of $setYaw$ actions can be found. This means that robustness is kind of built-in into the model, because discrepancies between the model and reality do not even occur. Unfortunately, nondeterminism very badly affects performance due to the high number of actions and the quadratically increasing number of interleaving combinations when concurrency is involved.

Gspandl et al. (2011) propose a way to handle sensed deviations of reality from the model by inserting additional actions. These actions restore consistency between the fluent values and reality. Whereas Figure 4.2 explains the deviations "in advance," the explaining actions are inserted after an inconsistency has been detected. This is, of course, much more suitable when it comes to managing belief online during program execution, but in the present scenario the upcoming observations are known in advance. Since each explaining action is still discrete, the whole approach boils down to the same number of actions like the robustness-by-nondeterminism approach.

Fuzzy fluents can be used to determine the degree by which some predicate holds. A fuzzy logic integration with the situation calculus has been proposed by Ferrein et al. (2008). However, at the lowest level the fluents still have quantitative values, the fuzzy predicates are stacked on these quantitative fluents. Therefore, when an observation is checked in a situation, from both directions qualitative values clash. Hence, fuzzy logic does not tackle the actual problem.

5.4.2 Probabilistic Extensions of the Situation Calculus and Golog

There are various probabilistic extensions of the situation calculus and Golog.

Bacchus et al. (1999) combine stochastic actions and a possible worlds approach to handle noise in the situation calculus. Stochastic actions have a nominal value set by the agent, and an actual value that is chosen by nature. When an action a is executed and s is among the situations currently considered possible, this leads to new situations $do(a', s)$ where a' is indistinguishable from a. For example, $move(3, y)$ with nominal value 3 and unknown actual value y might have indistinguishable outcomes $move(3, 2)$ and $move(3, 4)$. An action likelihood function $l(a, s)$ yields the probability for a being executed in s. Each situation in the initial epistemic state is assigned a weight, and while the epistemic state emerges, the weight is updated considering the action's likelihood. The probability that a formula ϕ holds in s is determined as the division of (a) the sum of the weights of the possible situations and (b) the sum of the weights of all situations considered possible. The point of the epistemic state is that sense action thin it out and thereby gain knowledge. The likelihood $l(a, s)$ being a fluent allows for a powerful context dependent error model. The formalization in (Bacchus et al., 1999) requires $l(a, s)$ to be a finite discrete probability distribution. Continuous distributions are simply discretized.

Mateus et al. (2001) propose a variant of the situation calculus that also supports continuous probabilistic actions. However, they do not axiomatize the integration and differentiation operators, whereas Bacchus et al. (1999) do formalize their sum. Furthermore, their approach really changes the situation calculus in that the primitive actions itself become nondeterministic.

Boutilier et al. (2000a,b) propose a decision-theoretic dialect of Golog called DTGolog. This approach is the only one of those presented which is not limited to the situation calculus but also defines the semantics of a Golog variant. In contrast to stGolog, (Reiter, 2001, Chapter 12), DTGolog also supports nondeterminism. DTGolog amalgamates Golog with fully observables Markov Decision Processes. The stochastic actions of (Boutilier et al., 2000a,b) are similar to those proposed by Bacchus et al. (1999), but it goes without the epistemic state. In DTGolog, the outcome actions represent the different transitions an action can induce in a Markov Decision Process. At nondeterministic choice points, it chooses the branch or value, respectively, that promises the highest reward value (roughly speaking).

The following distinguishes between nondeterminism and stochastic actions. Even though stochastic actions are kind of inherently nondeterministic, this is not what is meant by nondeterminism. Stochastic actions mean actions where nature chooses the outcome, whereas nondeterminism allows the agent to choose. Examples for nondeterministic features are the pick operator $\pi v \,.\, \sigma$ and branching $\sigma_1 \mid \sigma_2$.

Reiter's (2001) stochastic actions and decision-theoretic Golog are based on Bacchus et al. (1999) and Boutilier et al. (2000a,b). The primitive actions $A_1(\vec{x}_1), \ldots, A_k(\vec{x}_k)$

that are potential outcomes of a stochastic action $B(\vec{x})$ are enumerated in a macro: $\mathit{choice}(B(\vec{x}), a) \overset{\text{def}}{=} a = A_1(\vec{x}_1) \vee \ldots \vee a = A_k(\vec{x}_k)$. Additionally, a macro $\mathit{prob}_0(\alpha, \beta, s)$ specifies the probability that nature chooses α as outcome of β in s. Using these macros, he defines the stochastic Golog dialect stGolog. stGolog only handles deterministic programs. Since at each execution of a stochastic action nature picks one randomly, the execution of a program yields a tree with situation terms and associated probabilities at the leaves. The probability that a formula ϕ holds after executing a program σ is the sum of the probabilities of all situations s in the respective tree for which $\phi[s]$ holds.

A different way to integrate probabilities and the situation calculus is described by Fritz and McIlraith (2009). The values of fluents are interpreted as random variables with associated probability distributions. For example, the price of some product might be normally distributed around some measured price μ with deviation σ increasing with time. The regressed goal formula then contains probability distribution terms, e.g., $f_{\mathcal{N}}(\mathit{price}, 13, 1) < 11$. They compute the probability that the goal formula holds considering the random variables. This gives what they call the plan's robustness.

5.4.3 Robustness through Stochastic Actions

As described above, nondeterminism is computationally very expensive. This makes robustness by measuring the deviation interesting. The modeler might want to express that the car should accelerate with "about $3\,\mathrm{m/s^2}$." Fuzzy logic (Ferrein et al., 2008) would lend itself to model an expression like "about $3\,\mathrm{m/s^2}$," but it appears difficult to value the deviation of measurements from $3\,\mathrm{m/s^2}$. Probability as quantified belief might be suited better. More precisely, the car's acceleration might be a random variable X with a Gaussian distribution with mean $\mu = 3\,\mathrm{m/s^2}$ and deviation $\sigma = 0.5\,\mathrm{m/s^2}$.

Random variables as proposed by Fritz and McIlraith (2009) can express exactly this. But for an observation $\mathit{accel} = 3.5\,\mathrm{m/s^2}$, there is no intuitive way to get a probability that this holds. Perhaps one would measure this by $1 - \Pr(X \geq 3.5\,\mathrm{m/s^2})$ or so. Another critical point is how a single random variable X could explain two different measurements, because X can take only one actual value.

Outline

Stochastic actions similar to (Bacchus et al., 1999; Mateus et al., 2001; Boutilier et al., 2000a,b; Reiter, 2001) can be used to overcome these limitations. Each stochastic action has potentially countably infinitely many outcomes. The driver, from the perspective of plan recognition being a *part of nature*, chooses one of these outcome actions. An action likelihood function $\mathit{prob}_0(\beta, \alpha, s)$ denotes the probability that, when stochastic action β is to be executed in s, nature chooses α as outcome.

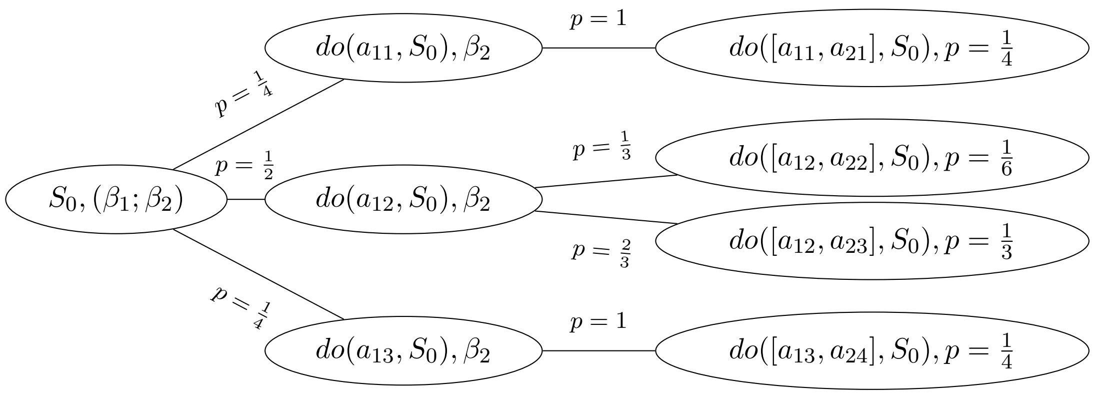

$S_0, (\beta_1; \beta_2)$
$p = \frac{1}{4}$
$p = \frac{1}{2}$
$p = \frac{1}{4}$
$do(a_{11}, S_0), \beta_2$
$do(a_{12}, S_0), \beta_2$
$do(a_{13}, S_0), \beta_2$
$p = 1$
$p = \frac{1}{3}$
$p = \frac{2}{3}$
$p = 1$
$do([a_{11}, a_{21}], S_0), p = \frac{1}{4}$
$do([a_{12}, a_{22}], S_0), p = \frac{1}{6}$
$do([a_{12}, a_{23}], S_0), p = \frac{1}{3}$
$do([a_{13}, a_{24}], S_0), p = \frac{1}{4}$

Figure 5.1: Tree of situation terms induced by executing two stochastic actions. β_1 has outcome actions a_{11}, a_{12}, a_{13} in S_0 with probabilities $\frac{1}{4}, \frac{1}{2}$ and $\frac{1}{4}$, respectively. When β_2 is executed in $do(a_{11}, S_0)$, nature may choose only a_{21}. In $do(a_{12}, S_0)$, nature may choose a_{22} or a_{23} with probabilities $\frac{1}{3}$ and $\frac{2}{3}$, respectively. In $do(a_{13}, S_0)$ nature picks a_{24}.

As in stGolog (Reiter, 2001), executing a deterministic program σ involving stochastic actions induces a tree with situation terms and their probabilities at the leaves. For each leaf situation term s holds $Do(\sigma, S_0, s)$ and its probability is the product of the probabilities of all outcome actions in s. Such a tree is depicted in Figure 5.1. Then, the probability that ϕ holds after executing σ is the sum of all leaf situations s for which $\phi[s]$ holds.

The modeler supplies macros $Choice(\beta, \alpha)$ and $prob_0(\beta, \alpha, s)$. $Choice(\beta, \alpha)$ indicates that α is a potential outcome action of stochastic action β and $prob_0(\beta, \alpha, s)$ is a function that returns the likelihood that nature chooses α as outcome of β in s. The action α is not time-stamped, because the time parameter is added at execution time. $prob_0$ being situation-dependent allows for a context-sensitive error model as in (Bacchus et al., 1999). The modeler is responsible to ensure that $prob_0(\beta, \alpha, s)$ defines a valid probability distribution. The distribution is assumed to be discrete; continuous distributions need to be discretized as in (Bacchus et al., 1999). The axiomatizer has to ensure that for any non-time-stamped primitive action α and stochastic action β

$$\mathcal{D} \models Choice(\beta, \alpha) \land (\exists \tau \,.\, \tau \geq start(s) \land Poss(\alpha[s, \tau], s)) \supset prob_0(\beta, \alpha, s) > 0$$

and

$$\mathcal{D} \models (\exists \alpha \,.\, Choice(\beta, \alpha) \land \exists \tau \,.\, \tau \geq start(s) \land Poss(\alpha[s, \tau], s)) \supset \sum_{\substack{\{\alpha \mid Choice(\beta,\alpha) \land \\ \exists \tau \,.\, \tau \geq start(s) \land \\ Poss(\alpha[s,\tau],s)\}}} prob_0(\beta, \alpha, s) = 1.$$

The sigma sign is axiomatized below in Formula (5.5) on page 45. It is important to note that $prob_0$ refers to the non-time-stamped actions. Besides this, both demands are equal to (12.2) and (12.3) by Reiter (2001, pp. 341f). The summation condition matches Reiter's *prob* macro, because it sums $prob_0$ of all possible outcome actions while it skips the impossible ones. Reiter's *prob* returns $prob_0$ for each possible and executable outcome action and returns 0 otherwise.

In contrast to (Reiter, 2001), this *Choice* allows an infinite number of outcome actions. This is relevant, because a discretely distributed random variable may have an infinite number of values with positive probability. Since the sum's formalization (5.5) requires the set under the $\sum$ symbol to be at most countably infinite, the number of outcome actions needs to be countable, too:

$$\mathcal{D} \models \forall \beta \,.\, \exists f \,.\, \forall \alpha \,.\, Choice(\beta, \alpha) \supset (\exists i) f(i) = \alpha.$$

Here, i is of sort natural numbers and f is a second-order function variable. This means that for each stochastic action β, there is a mapping from the natural numbers to the outcome actions.

The new sort of probabilities is assumed to be the set of real numbers with their standard interpretation.

An approach alternative to probabilities would be to assign weights to situations. As done by Bacchus et al. (1999), weights can then be normalized to 1.

Similar to DTGolog (Boutilier et al., 2000a,b), nondeterminism in programs is resolved by choosing the branch which maximizes the probability that some situation- and time-suppressed goal formula holds after executing the program. In the field of plan recognition, this formula might express that all observations are entailed by the current situation.

In fact, DTGolog does not optimize with respect to a goal formula, but maximizes the value of a reward function. While in DTGolog, rewards are added up along the path of execution, this Golog dialect will only consider a single situation's reward (see definition of *value* (5.4) below). Hence, optimizing the probability that goal formula ϕ holds can thus be captured by the simple reward function

$$r(s) = \begin{cases} 1 & \text{if } \phi[s] \\ 0 & \text{else.} \end{cases} \tag{5.3}$$

In the following, this is called the standard reward function if ϕ holds if all observations are entailed. Of course, ϕ has to be known in advance in order to resolve nondeterminism that way. Like in DTGolog, optimization could be done modulo a certain horizon to improve performance.

The concept of reward functions is superior to logical goal formulas, because it allows to rate situations not only as "good" and "bad" but also allows statements in between.

This comes in handy when not all observations are known in advance and/or the search horizon is limited. In realistic scenarios, the set of observations grows as time goes by. The reward function could map each situation to the number of entailed observations. Then, the candidate program could be executed incrementally corresponding to the number of occurred observations and the reward function would guide the interpreter a reasonable way through the program. This is explored in detail in the next chapter.

A difference to DTGolog is that DTGolog uses evaluation semantics and defines *BestDo*, a variant of *Do* that resolves nondeterminism. To preserve the support of concurrency, the ideas of *BestDo* must be transferred to *Trans*. Fritz and McIlraith (2006) propose a semantics using *BestTrans*, but their goal is just synchronous execution of programs and not concurrency. Furthermore, they use lists for bookkeeping of branching decisions. Lists, however, are not a typical construct used in logic.

To incorporate the probabilities, the predicate *Trans* is replaced with a function *transPr* similar as in pGolog (Grosskreutz, 2000; Grosskreutz and Lakemeyer, 2000b). The probabilities in pGolog stem from choice points augmented with the likelihood of each branch. In pGolog, $transPr(\sigma, s, \delta, s') = p$ holds if σ executed in s leads to s' with remaining program δ with probability p. Where *Trans* fails, *transPr* returns probability 0.

Since nondeterminism is resolved by choosing the best branch, an additional parameter r for *transPr* is needed for the reward function to value branches. The semantics is defined by the function

$$transPr(r, \sigma, s, \delta, s') = p$$

which holds if

- r denotes a functional fluent that returns some non-negative reward for each situation,[1]
- a single transition – that is, the next primitive, stochastic or test action – of σ in s leads to s',
- the rest of σ after this transition is δ,
- nondeterminism in σ is resolved by inspecting all possible executions and choosing the branch that maximizes the expected reward (for the special case of the standard reward function (5.3) this means: the nondeterminism in σ is resolved by branching in a way that maximizes the probability that the observations are entailed after executing σ), and
- p is the probability of getting to s' by a single transition of σ in s (for non-stochastic actions, this is either 0 or 1).

Before *transPr* can be defined, a number of helper macros and predicates have to be introduced. These include

[1]For readability, reward functions are given and used in normal mathematical style in the following.

- *value* to compute the estimated reward value after executing a given program in a given situation,
- the *Do* and $Trans^*$ equivalents *doPr* and $transPr^*$,
- *Next* and *MaybeFinal* that are used to decompose a program into an atomic (that is, primitive, stochastic or test) action and a remainder,
- *transAtPr* to execute an atomic action, and finally
- *transPr* and *Final* that mimic the behavior of *Trans* and *Final* in the traditional transition semantics.

Rating Situations with *value*

There are many ways to combine the rewards of a situation and its ancestor situations. For example, DTGolog adds up the rewards of all situations. In the present domain, the intention is rather that there should be one situation that explains all observations. Hence, the value of a branch should be the maximum encountered reward along this path. However, by modifying the reward functions, one can reproduce either behavior.

Figure 5.2 visualizes the potential outcomes of executing a deterministic program consisting of a sequence of two stochastic actions. For example, if outcome action a_{11} occurs, the reward of 1 is worse than the average reward of the children $\frac{1}{2} \cdot 3 + \frac{1}{2} \cdot 2$. Similarly, the children of $do(a_{12}, S_0)$ have a higher average reward than 4 even though one child is worse than $do(a_{12}, S_0)$. In the third branch, $do(a_{13}, S_0)$'s reward of 4 is better than its children's average of 2. Note that the initial situation has a reward of $3\frac{1}{3}$, but it does not win even though its children have an average reward of only 3 and its grandchildren have an average reward of $3\frac{1}{6}$. This is because the descendant situations marked with the double ellipses have a better average reward of $\frac{1}{3} \cdot (\frac{1}{2} \cdot (3 + 2 + 10 + 0) + 4) = 3\frac{5}{6}$. Hence, the function *value* should pick these situations to determine the estimated reward.

A set S of situations from a situation tree is called a *path cover* if, for any path from the root to a leaf in the situation tree, there is one $s \in S$ that lies on the path. Obviously, a single situation can cover multiple paths, but no path may have two of its situations in the path cover. A path cover is called *optimal* if the average reward of its situations is greater or equal than any other path cover's. The root node is always a trivial path cover. In Figure 5.2, the nodes with double ellipses are an optimal path cover.

Consider a situation tree induced by executing σ in s. The following macro asserts that the path cover $\{s\}$ is optimal, that is, no path cover has a higher estimated reward

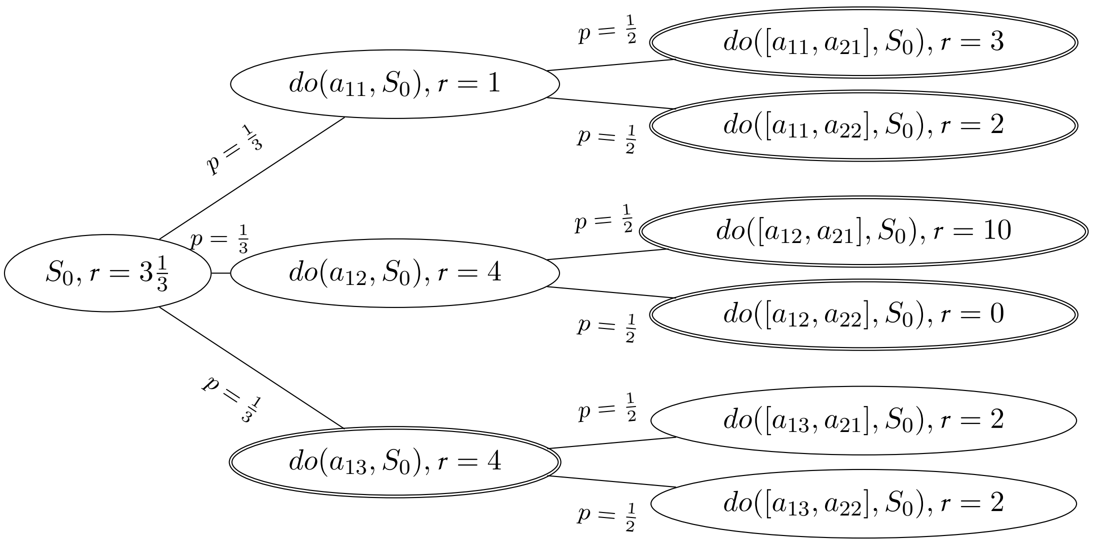


Figure 5.2: Value determination of two sequential stochastic actions. Each node represents one of the outcomes of a stochastic action. It is labelled with the reached situation's reward. The edges' labels are the probabilities of the respective outcomes. The double ellipses mark those situations that maximize the average reward.

than the root node:

$$Best(r, \sigma, s) \overset{\mathsf{def}}{=} \forall P \,.\, \big(\forall s', s'' \,.\, P(s') \wedge P(s'') \supset s' \not\sqsubset s''\big) \supset \sum_{\substack{\{(p,s') \,|\, \exists\delta \,.\, transPr^*(r,\sigma,s,\delta,s')=p \,\wedge \\ p>0 \,\wedge\, P(s')\}}} p \cdot r(s') \leq r(s).$$

The sigma sign is axiomatized below; the summation iterates over all values for p and s' that make the sum's condition true. Notice that the set of (p, s') is not uncountable even though probabilities and situations are. This is because there are only countably many transitions of σ and at each transition, at most countably infinitely many outcome actions yield a positive return value of $transPr$. The second-order variable P represents an arbitrary set of situations which are pairwise non-descendants/ancestors, or in other words, for all paths from from s to a leaf situation, at most one situation from this path is contained in P. For all situations which are not reachable from s by σ, $p = 0$ holds and therefore the addends have no effect. Hence, the situations in P with a positive probability form either a path cover or a subset of a path cover. This captures that in Figure 5.2, S_0 is better than its children, and it is also better than its grandchildren, but there still is a path cover that is even better than S_0: considering the situations with double ellipses, each one is from a different branch and therefore they form a valid path cover, and their average reward is better than S_0's. Recall that $transPr^*$ resolves nondeterminism in σ, hence, the situations with $p > 0$ stem only from different outcome actions and not from nondeterministic program constructs.

What *value* still needs to do is taking the *earliest* situation which maximizes the estimated reward. Using *Best*, the function *value* can be defined as

$$value(r,\sigma,s) \overset{\text{def}}{=} \sum_{\substack{\{(p,s') \mid \exists\delta\,.\,transPr^*(r,\sigma,s,\delta,s')=p \,\wedge \\ p>0 \,\wedge\, Best(r,\delta,s') \,\wedge \\ \neg\exists s'',\delta\,.\,transPr^*(r,\sigma,s,\delta,s'')>0 \,\wedge \\ Best(r,\delta,s'') \,\wedge\, s'' \sqsubset s'\}}} p \cdot r(s'). \tag{5.4}$$

This determines the maximum reward of each branch and calculates the average. In terms of Figure 5.2, the first *Best* in the sum's condition holds for situations whose reward is better than its descendants'. For example, it succeeds for $do(a_{13}, S_0)$, but also for its children $do([a_{13}, a_{21}], S_0)$ and $do([a_{13}, a_{22}], S_0)$. The second *Best* rules out these children. Therefore, the whole definition of *value* sums the reward values multiplied with the probabilities. Again, the set under the sigma sign is countable for the same reasons as with the definition of *Best* above. Note that in contrast to DTGolog, *value* does not take into account whether or not a branch is executable in total, that is, leads to a final configuration.

The sigma signs in *Best* and *value* still need to be axiomatized. The following definitions refers to the general case

$$\sum_{\{\vec{x} \mid \Phi[\vec{X}/\vec{x}]\}} \nu(\vec{x})$$

where the set under the sigma sign $\{\vec{x} \mid \Phi[\vec{X}/\vec{x}]\}$ is countable and all addends are non-negative real numbers: $\nu(\vec{x}) \geq 0$. Φ mentions some constants $X_1, \ldots, X_n$ which are taken together in $\vec{X}$ and substituted by $\vec{x}$. For the sake of readability, the constants $\vec{X}$ and the variables $\vec{x}$ had the same names in the above applications, and the substitution was omitted. The sum can be formalized in second-order logic similar to (Bacchus et al., 1999):

$$\begin{aligned} sum_\nu(\Phi(\vec{X})) = v \overset{\text{def}}{=} \exists f,g\,.\,&(\forall\vec{x})\,\big(\Phi[\vec{X}/\vec{x}] \supset (\exists i)\vec{x} = g(i)\big) \,\wedge \\ &(\forall i,j)\,\big(\Phi[\vec{X}/g(i)] \wedge \Phi[\vec{X}/g(j)] \wedge i \neq j \supset g(i) \neq g(j)\big) \,\wedge \\ &f(0) = 0 \,\wedge \\ &(\forall i)\,\big((\Phi[\vec{X}/g(i)] \supset f(i+1) = f(i) + \nu(g(i))) \,\wedge \\ &\qquad(\neg\Phi[\vec{X}/g(i)] \supset f(i+1) = f(i))\big) \,\wedge \\ &(\forall i)\,\big(f(i) \leq v \,\wedge \\ &\qquad(\forall v')(f(i) \leq v' \supset v \leq v')\big). \end{aligned} \tag{5.5}$$

All variables i and j are supposed to be natural numbers, which can be easily defined with 0, 1 and + (Bacchus et al., 1999). The second-order function g enumerates all vectors $\vec{x}$ for which the sum's condition Φ holds and $f(i+1)$ sums the values $\nu(\vec{x})$ of all vectors $g(0), \ldots, g(i)$. Note that g "contains" each $\vec{x}$ for which $\Phi[\vec{X}/\vec{x}]$ holds. Other values of g are not defined, which is not critical, because they are not referenced. Since all values of g are distinct, each $\vec{x}$ with $\Phi[\vec{X}/\vec{x}]$ is counted only once. The last two lines assert that v is the least upper bound of f. The *sum* macro is only satisfiable if the sum converges.

The restriction to converging series is actually relevant. For example, $value(r, \alpha^*, s)$ is undefined if the reward function r increases with each α. It is the axiomatizer's job to ensure that the combination of reward function and programs has no such effect. Alternatively, one might drop the nondeterministic iteration (and thereby, *while* loops, which are defined in terms of nondeterministic iteration and tests) and recursive procedure calls. However, there are realistic scenarios in which actions in a nondeterministic loop have no effect on the reward at all.

Reflexive Closure $transPr^*$ and $doPr$

To ease the definition of functions, the following macro is frequently used in the following:

$$\textbf{if } \xi \textbf{ then } \phi \textbf{ else } \psi \overset{\text{def}}{=} \xi \wedge \phi \vee \neg\xi \wedge \psi.$$

Variables quantified in ξ are also intended to be visible in ϕ. Sometimes, a special quantifier, $\exists^1$, is used inside ξ. The idea is that the condition might hold for several different values of the existentially quantified variables, but to ensure determinism, the condition should succeed for only a single value for each variable. This behavior can be defined as

$$\begin{aligned}&\textbf{if } \chi \wedge \exists^1 \vec{x} \,.\, \xi \textbf{ then } \phi \textbf{ else } \psi \overset{\text{def}}{=}\\ &\quad \textbf{if } \chi \wedge \exists \vec{x} \,.\, \xi \wedge \big(\forall \vec{x}' \,.\, \xi^{\vec{x}}_{\vec{x}'} \supset \vec{x} \leq \vec{x}'\big) \textbf{ then } \phi \textbf{ else } \psi\end{aligned}$$

where $\vec{x}'$ is a vector of new variables that do not occur in ξ and $\leq$ is some total ordering, e.g., the lexicographical one. χ is optional.

The reflexive transitive closure of $transPr$ is second-order defined as

$$\begin{aligned}&transPr^*(r, \sigma, s, \delta, s') = p \overset{\text{def}}{=}\\ &\textbf{if } \exists p' \,.\, \forall f \,.\, \big(\forall r', \sigma_1, s_0 \,.\, f(r', \sigma_1, s_0, \sigma_1, s_0) = 1\big) \wedge\\ &\qquad\qquad \big(\forall r', \sigma_1, \delta_1, \delta_2, s_0, s_1, s_2, p_1, p_2 \,.\\ &\qquad\qquad\quad p_1 > 0 \wedge f(r', \sigma_1, s_0, \delta_1, s_1) = p_1 \wedge\\ &\qquad\qquad\quad p_2 > 0 \wedge transPr(r', \delta_1, s_1, \delta_2, s_2) = p_2 \supset\\ &\qquad\qquad\qquad f(r', \sigma_1, s_0, \delta_2, s_2) = p_1 \cdot p_2\big) \supset\\ &\qquad\qquad f(r, \sigma, s, \delta, s') = p'\\ &\textbf{then } p = p' \textbf{ else } p = 0.\end{aligned} \tag{5.6}$$

This definition is analogous to $transPr^*$ in (Grosskreutz, 2000; Grosskreutz and Lakemeyer, 2000b). The macro $doPr$ uses $transPr^*$ to execute the program, but stops at the first point it is final:

$$\begin{aligned}
&doPr(r, \sigma, s, s') = p \overset{\text{def}}{=} \\
&\quad \textbf{if } \exists p' \,.\, transPr^*(r, \sigma, s, s') = p' \wedge Final(r, \sigma, s') \wedge \\
&\qquad (\forall s'')\big(s \sqsubseteq s'' \wedge s'' \sqsubset s' \supset \neg Final(r, \sigma, s'')\big) \\
&\quad \textbf{then } p = p' \textbf{ else } p = 0.
\end{aligned}$$

Decomposing Programs with $Next$ and $MaybeFinal$

The $Next$ predicate is very similar to the $Trans$ predicate in the standard transition semantics. $Next(\sigma, \gamma, \delta)$ holds if there is an execution of σ such that the next atomic action is γ and the remaining program is δ. Thus, in contrast to $Trans$, $Next$ does not actually execute actions and create new situations. Instead, it breaks down the program up to the level of primitive, stochastic and test actions.

$$\begin{aligned}
Next(Nil, \gamma, \delta) &\equiv False \\
Next(\alpha, \gamma, \delta) &\equiv \gamma = \alpha \wedge \delta = Nil \\
Next(\beta, \gamma, \delta) &\equiv \gamma = \beta \wedge \delta = Nil \\
Next(\phi?, \gamma, \delta) &\equiv \gamma = \phi? \wedge \delta = Nil \\
Next(\pi v \,.\, \sigma, \gamma, \delta) &\equiv \exists x \,.\, Next(\sigma^v_x, \gamma, \delta) \\
Next(\sigma_1 \,|\, \sigma_2, \gamma, \delta) &\equiv Next(\sigma_1, \gamma, \delta) \vee Next(\sigma_2, \gamma, \delta) \\
Next(\sigma_1; \sigma_2, \gamma, \delta) &\equiv \exists \sigma_1' \,.\, Next(\sigma_1, \gamma, \sigma_1') \wedge \delta = \sigma_1'; \sigma_2 \vee \\
&\qquad MaybeFinal(\sigma_1) \wedge Next(\sigma_2, \gamma, \delta) \\
Next(\sigma_1 \parallel \sigma_2, \gamma, \delta) &\equiv \exists \sigma_1' \,.\, Next(\sigma_1, \gamma, \sigma_1') \wedge \delta = \sigma_1' \parallel \sigma_2 \vee \\
&\qquad \exists \sigma_2' \,.\, Next(\sigma_2, \gamma, \sigma_2') \wedge \delta = \sigma_1 \parallel \sigma_2' \\
Next(\sigma^*, \gamma, \delta) &\equiv \exists \sigma' \,.\, Next(\sigma, \gamma, \sigma') \wedge \delta = \sigma'; \sigma^*.
\end{aligned} \tag{5.7}$$

Recall that α stands for a single non-time-stamped primitive action and β denotes a single stochastic action. In $Next$ and in the following, γ stands for an action that is either non-time-stamped primitive, stochastic or a test.

$MaybeFinal(\sigma)$ is intended to succeed if there is a way to branch in the possibly non-deterministic program σ that is final, hence the "maybe." The definition is essentially the same as $Final$ in the standard transition semantics:

$$\begin{aligned}
MaybeFinal(Nil) &\equiv True \\
MaybeFinal(\alpha) &\equiv False \\
MaybeFinal(\beta) &\equiv False \\
MaybeFinal(\phi?) &\equiv False
\end{aligned}$$

$$
\begin{aligned}
MaybeFinal(\pi v\,.\,\sigma) &\equiv \exists x\,.\,MaybeFinal(\sigma^v_x)\\
MaybeFinal(\sigma_1 \,|\, \sigma_2) &\equiv MaybeFinal(\sigma_1) \vee MaybeFinal(\sigma_2)\\
MaybeFinal(\sigma_1;\sigma_2) &\equiv MaybeFinal(\sigma_1) \wedge MaybeFinal(\sigma_2)\\
MaybeFinal(\sigma_1 \parallel \sigma_2) &\equiv MaybeFinal(\sigma_1) \wedge MaybeFinal(\sigma_2)\\
MaybeFinal(\sigma^*) &\equiv True.
\end{aligned}
$$

The motivation behind $Next$ and $MaybeFinal$ is the following: $transPr$ picks that one out of all possible decompositions $(\gamma;\delta)$ which maximizes the reward after its execution where γ is a single atomic action. This property is crucial to handle concurrency, because it allows to steadily push back the concurrency in σ to δ until δ is Nil.

In contrast to original ConGolog's $Final$, $MaybeFinal$ has no parameter for the situation term. This is because we go without synchronized conditional statements and loops for simplicity (cf. Section 3.2). Furthermore, this definition of $Next$ and $MaybeFinal$ does not include recursive procedures. However, this feature can be integrated the same way as in ConGolog (Giacomo et al., 2000).

Stepwise Execution with $transAtPr$, $transPr$ and $Final$

In comparison to the standard $Trans$, execution of atomic actions is moved to a separate function called $transAtPr$. $transAtPr(r,\gamma,\delta,s,s') = p$ is read as

- the primitive, stochastic or test action γ leads from s to s',
- the timestamp of an executed primitive action maximizes the reward r after executing δ in s',
- for primitive and test actions, $p = 1$ and $p = 0$ indicate success and failure, respectively; for stochastic actions, p is the probability that the outcome action reflected in s' is chosen and executed successfully.

The function is defined as follows:

$$
\begin{aligned}
&transAtPr(r,\alpha,\delta,s,s') = p \equiv \qquad (5.8)\\
&\quad \textbf{if } \exists^1\tau\,.\,\tau \geq start(s) \wedge Poss(\alpha[s,\tau],s) \wedge s' = do(\alpha[s,\tau],s) \wedge\\
&\qquad \big(\forall\tau',s''\,.\,\tau' \geq start(s) \wedge Poss(\alpha[s,\tau'],s) \wedge s'' = do(\alpha[s,\tau'],s) \supset\\
&\qquad\quad value(r,\delta,s') \geq value(r,\delta,s'')\big)\\
&\quad \textbf{then } p = 1 \textbf{ else } p = 0
\end{aligned}
$$

$$
\begin{aligned}
&transAtPr(r,\beta,\delta,s,s') = p \equiv \qquad (5.9)\\
&\quad \textbf{if } \exists\alpha,p'\,.\,Choice(\beta,\alpha) \wedge transAtPr(r,\alpha,\delta,s,s') \cdot prob_0(\beta,\alpha,s) = p' \wedge p' > 0\\
&\quad \textbf{then } p = p' \textbf{ else } p = 0
\end{aligned}
$$

$$
\begin{aligned}
&transAtPr(r,\phi?,\delta,s,s') = p \equiv \qquad (5.10)\\
&\quad \textbf{if } \phi[s] \wedge s' = s \textbf{ then } p = 1 \textbf{ else } p = 0.
\end{aligned}
$$

With all these tools in hand, $transPr(r, \sigma, \delta, s')$ simply needs to choose that partition $(\gamma; \delta)$ of σ which maximizes the reward after executing the total program. The resulting definition of $transPr$ is concise:

$$\begin{aligned}
&transPr(r, \sigma, s, \delta, s') = p \equiv \qquad (5.11)\\
&\quad \textbf{if } \exists^1 \gamma_1, \delta_1 \,.\, Next(\sigma, \gamma_1, \delta_1) \land\\
&\qquad \big(\forall \gamma_2, \delta_2 \,.\, Next(\sigma, \gamma_2, \delta_2) \supset value(r, (\gamma_1; \delta_1), s) \geq value(r, (\gamma_2; \delta_2), s)\big)\\
&\quad \textbf{then } \big(\textbf{if } \delta = \delta_1 \textbf{ then } p = transAtPr(r, \gamma_1, \delta_1, s, s') \textbf{ else } p = 0\big)\\
&\quad \textbf{else } p = 0.
\end{aligned}$$

A configuration is considered final if there is a final branching of σ and all ongoing executions of σ do not yield a higher reward:

$$Final(r, \sigma, s) \equiv MaybeFinal(\sigma) \land value(r, Nil, s) \geq value(r, \sigma, s). \qquad (5.12)$$

Why Program Decomposition is Crucial

Why is it necessary to introduce $Next$ instead of fusing DTGolog-style resolving of nondeterminism into the $Trans$ predicate? The reason is concurrency. As shown in rule (5.2) for $\sigma_1 \parallel \sigma_2$, one of the concurrent subprograms, say σ_1, is chosen nondeterministically and is then transitioned. σ_1 itself may be nondeterministic, too. For example, one might have $\sigma_1 = ((\sigma_{1,1} \,|\, \sigma_{1,2}); \sigma_1')$. Then, to decide whether to branch to $\sigma_{1,2}$ or $\sigma_{1,2}$, the interpreter needs to determine whether $((\sigma_{1,1}; \sigma_1') \parallel \sigma_2)$ or $((\sigma_{1,2}; \sigma_1') \parallel \sigma_2)$ is the better choice.

Keeping track of σ_2 as the concurrently running program does not solve the problem, because $\sigma_{1,1}$ and $\sigma_{1,2}$ might contain further nested concurrency.

With $Next$, $\sigma_1 \parallel \sigma_2$ is decomposed in all possible ways. One decompositions is $(\gamma; ((\sigma_{1,1}'; \sigma_1') \parallel \sigma_2))$, that is, $\sigma_{1,1}$ is split into γ and $\sigma_{1,1}'$. The analogous decomposition for the case that the interpreter branches to $\sigma_{1,2}$ is $(\gamma; ((\sigma_{1,2}'; \sigma_1') \parallel \sigma_2))$ where $\sigma_{1,2}$ itself is decomposed into γ and $\sigma_{1,2}'$. Further decompositions exist for the case that the interpreter actually transitions σ_2 before σ_1.

This behavior cannot be achieved with classic $Trans$, because it recursively disassembles the program following its abstract syntax tree. During this, $Trans$ does not keep track of the remaining program, which would be needed to resolve nondeterminism.

The described procedure continuously pushes back the concurrency operator: for some program $(\alpha_1; \alpha_3) \parallel \alpha_2$, the interpreter might choose the decomposition into α_1 and $\alpha_3 \parallel \alpha_2$. Then, the interpreter might pick α_2 which means the remainder is $\alpha_3 \parallel Nil$. Finally, α_3 is transitioned and the remainder $Nil \parallel Nil$ is final.

Well-Definedness of the Semantics

In the following, some properties of this semantics are proven. To begin with, it is shown that the semantics is well-defined. From now on, $\mathcal{C}$ denotes the axiomatization of the new *transPr* semantics.

Furthermore, we assume in all proofs that r and σ (and other executed programs) do not lead to a loop with ever-increasing reward. Otherwise, the sums in *value* and *Best* were undefined as argued above.

Theorem 5.1. *transPr returns a new configuration for each input configuration:*

$$\forall r, \sigma, s \,.\, \exists \delta, s', p \,.\, transPr(r, \sigma, s, \delta, s') = p.$$

Proof. We show that for any input configuration, we can determine an output configuration by simply following the definition of *transPr* and its helpers.

transPr leads to recursive evaluations of itself via *transAtPr* and *value*. One needs to show that at each recursion step, the programs shrink and eventually are *Nil* at which point the recursion ends, or that the literals that recursively rely on *transPr* (such as *value* comparisons) are otherwise decidable.

- Consider the subformula

  $$\forall \gamma_2, \delta_2 \,.\, Next(\sigma, \gamma_2, \delta_2) \supset value(r, (\gamma_1; \delta_1), s) \geq value(r, (\gamma_2; \delta_2), s)$$

  of *transPr* (5.11). In the following, superscript n distinguishes the variables at depth of recursion n from previous and subsequent recursion steps of *transPr*. Let $(\gamma_i^n; \delta_i^n)$, $i \in \{1, 2\}$ be two decompositions of σ^n in *transPr*. *value* uses *transPr** which leads to a recursive evaluation of *transPr* for $\sigma^{n+1} = (\gamma_i^n; \delta_i^n)$ (or real subprograms). The only decomposition of σ^{n+1} is simply $(\gamma_i^{n+1}; \delta_i^{n+1}) = (\gamma_i^n; \delta_i^n)$, because γ_i^n is an atomic action. Therefore, the literal

  $$value(r, (\gamma_1^{n+1}; \delta_1^{n+1}), s) \geq value(r, (\gamma_2^{n+1}; \delta_2^{n+1}), s)$$

  is trivially true.

- After *transPr* has chosen a decomposition $(\gamma_1^n; \delta_1^n)$ of σ^n, it evaluates *transAtPr* to execute γ_1^n taking account of the remaining program δ_1^n. *transAtPr* (5.8) then proceeds with

  $$value(r, \delta_1^n, s') \geq value(r, \delta_1^n, s'')$$

  which finally leads to recursive evaluations of *transPr*. The program which *transPr* executes at recursion step $n + 1$ is a subprogram of the program at step n: $\sigma^{n+1} = \delta_1^n$. For sufficiently great n, it is $\sigma^{n+1} = \delta_1^n = Nil$. At this point, there is no decomposition of σ^{n+1} and the recursion ends.

This procedure leads to a successor configuration for any input configuration. □

Corollary 5.2. *$transPr^*$ and $doPr$ return a successor configuration for each input configuration.*

Proof. Follows from Lemma 5.1. □

Lemma 5.3. *$transAtPr$ is a well-defined function:*

$$\mathcal{D} \cup \mathcal{C} \models transAtPr(r, \gamma, \delta, s, s') = p_1 \wedge transAtPr(r, \gamma, \delta, s, s') = p_2 \supset p_1 = p_2.$$

Proof. The rules (5.8) and (5.10) are both functions, because all variables mentioned in the condition of the **if-then-else** construct are either free or quantified with $\exists^1$. (5.9) is a function, because the outcome action of the stochastic action is uniquely determined by s and s'. □

Theorem 5.4. *$transPr$ is a well-defined function:*

$$\mathcal{D} \cup \mathcal{C} \models transPr(r, \sigma, s, \delta, s') = p_1 \wedge transPr(r, \sigma, s, \delta, s') = p_2 \supset p_1 = p_2.$$

Proof. Due to the $\exists^1$ quantor in the outer **if-then-else** construct, the decomposition $(\gamma_1; \delta_1)$ in (5.11) is unique. And since $transAtPr$ is a well-defined function according to Lemma 5.3, $transPr$ is so, too. □

Theorem 5.5. *$transPr^*$ is a well-defined function:*

$$\mathcal{D} \cup \mathcal{C} \models transPr^*(r, \sigma, s, \delta, s') = p_1 \wedge transPr^*(r, \sigma, s, \delta, s') = p_2 \supset p_1 = p_2.$$

Proof. To prove the claim, one needs to prove that f in (5.6) is a function. This can be shown by induction. Fixate r', σ_1 and s_0.

Base step. Let $k = 0$. $f(r', \sigma_1, s_0, \sigma_1, s_0) = 1$ is a well-defined function that applies $transPr$ up to k times to σ_1 in s_0. Furthermore, for those resulting situations s_1 that are the result of applying $transPr$ exactly k times, no super-situation $s_1' \sqsupset s_1$ is reached by f.

Inductive step. Assume $f(r', \sigma_1, s_0, \delta_1, s_1) = p_1 > 0$ is a well-defined function that applies $transPr$ up to k times. Furthermore, assume that for all resulting situations s_1 there are no super-situations $s_1' \sqsupset s_1$ reached by f.

According to Theorem 5.4, $transPr$ itself is a well-defined function. One needs to show that f is still a well-defined function if mappings

$$f(r', \sigma_1, s_0, \delta_2, s_2) = p_1 \cdot p_2$$

are added for all $transPr(r', \delta_1, s_1, \delta_2, s_2) = p_2 > 0$. Particularly, this means that each argument tuple of f uniquely determines the result. To distinguish the original and the modified f, the former is called f_1 and the latter is denoted by f_2 in the following.

- If δ_1, s_1 is the result of less than k applications of *transPr*, then f_1 already covers what happens when *transPr* is applied to δ_1, s_1.
- Assume δ_1, s_1 is the result of exactly k applications of *transPr*. There are three different cases:
  - If *transPr* chooses a test action, the result of *transPr* is is $\delta_2 = \delta$ and $s_2 = s_1$. Even though this mapping is already covered by f_1, there is no conflict, because $p_1 \cdot p_2 = p_1$. This is because for test actions, p_2 is either 0 or 1 and the first case is ruled out by requiring $p_2 > 0$.

    In this case, there is no other result of *transPr* besides $\delta_2 = \delta$ and $s_2 = s_1$. Thus, there is no super-situation $s_2' \sqsupset s_2$ reached by f_2.
  - If *transPr* performs a primitive action, say a, the new situation is $s_2 = do(a, s_1)$. Since there is no super-situation $s_1' \sqsupset s_1$ reached by f_1, particularly s_2 is not reached by f_1.

    Since the result of executing a primitive action is unambiguous, there are no other resulting situations except s_2 and therefore no super-situations of s_2 are reached by f_2.
  - If *transPr* executes a stochastic action, this potentially results in different s_2. However, they are pairwise incomparable, that is, for two outcome situations s_2 and s_2' holds $s_2 \not\sqsubset s_2'$ and $s_2' \not\sqsubset s_2$. For each s_2, the same reasoning as for primitive action applies.

The induction proves that f in (5.6) is a function. Due to the **if-then-else** construct, $transPr^*$ itself is a well-defined function, too. □

Trans is a Special Case of the *transPr* Semantics

The goal of the next theorems is to show that the new semantics includes the *Trans* semantics for non-stochastic programs as special case. To show relationships between the new *transPr* semantics and the old *Trans* semantics, both axiom sets are needed. Since both axiomatizations define distinct symbols, the union $\mathcal{C} \cup \mathcal{C}'$ can be used, where $\mathcal{C}'$ stands for the old transition semantics.[2]

Lemma 5.6. *If there is a transition in the nondeterministic interpreter, the program can also be partitioned:*

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models (\exists s')Trans(\sigma, s, \delta, s') \supset (\exists \gamma)Next(\sigma, \gamma, \delta).$$

Proof. The proposition holds because the functionality of *Next* is basically a subset of *Trans*. □

[2]Strictly speaking, this is not correct, because $\mathcal{C}'$ defines the binary relation *Final* and $\mathcal{C}$ defines a ternary *Final*. We assume an appropriate renaming to ensure uniqueness of the relation symbols.

Lemma 5.7. *If a transition with respect to $Next$ and $transAtPr$ of non-stochastic σ in s leads to remaining program δ in s', $Trans$ allows a transition of σ in s to the same remaining program δ and the same new situation s':*

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models Next(\sigma, \gamma, \delta) \wedge transAtPr(r, \gamma, \delta, s, s') > 0 \supset Trans(\sigma, s, \delta, s').$$

Proof. γ is is an atomic action, therefore $transAtPr$ being positive implies that γ is executable in s. This implies that $Trans$ can execute γ with the same new situation s':

- If γ is a primitive action, then $Trans$ (5.1) can choose the same timestamp as $transAtPr$ (5.8).
- γ cannot be a stochastic action, because σ is assumed to be non-stochastic.
- For a test action γ, the resulting situation s' is the same as s in both, $Trans$ (3.1) and $transAtPr$ (5.10).

And since γ is a first action of σ according to $Next$, $Trans$ considers executing γ. □

The reverse direction of Lemma 5.7 does not hold, because the executed action's timestamp that is valid in $Trans$ is not necessarily optimal in $transAtPr$. However, a relaxed version holds:

Lemma 5.8. *If a transition with respect to $Trans$ of σ in s leads to a remaining program δ in some situation, $Next$ and $transAtPr$ allow a transition of σ in s to the same remaining program δ:*

$$\begin{aligned} \mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models (\exists s')Trans(\sigma, s, \delta, s') \supset \\ \exists \gamma \,.\, Next(\sigma, \gamma, \delta) \wedge (\exists s')transAtPr(r, \gamma, \delta, s, s') > 0. \end{aligned}$$

Proof. The first part of the conclusion, the existence of a partition, follows from Lemma 5.6. And if γ is possible in $Trans$, it is also possible in $transAtPr$:

- If γ is a primitive action and $Trans$ (5.1) can execute it, there are one or multiple points in time at which γ's precondition holds. In particular, one of these points in time maximizes the reward after executing $(\gamma; \delta)$. $transAtPr$ (5.8) picks this point in time.
- If γ is stochastic, $Trans$ cannot succeed.
- For a test action γ, the resulting situation s' is equal to s in both, $Trans$ (3.1) and $transAtPr$ (5.10).

Therefore, $transAtPr$ is positive. □

Theorem 5.9. *A non-stochastic program σ can be transitioned with respect to $transPr$ if and only if it can be transitioned with respect to $Trans$:*

$$\begin{aligned}\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models\ & (\exists \delta, s', p)(transPr(r, \sigma, s, \delta, s') = p \wedge (p > 0 \vee r(s') = 0)) \subset \\ & (\exists \delta, s')Trans(\sigma, s, \delta, s').\end{aligned} \tag{5.13}$$

The $\supset$-direction of the equality is even stronger, because the result of transitioning a program with respect to $transPr$ is also a valid result with respect to $Trans$:

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models transPr(r, \sigma, s, \delta, s') > 0 \supset Trans(\sigma, s, \delta, s'). \tag{5.14}$$

Proof. The $\subset$-direction (5.13): Assume $(\exists \delta, s')Trans(\sigma, s, \delta, s')$ holds. Due to Lemma 5.8 there is a decomposition $(\gamma; \delta)$ such that γ can be executed by $transAtPr$. In particular, there is a decomposition that maximizes the estimated reward after executing σ in total. Let s' be the resulting situation of executing γ with $transAtPr$. Assume $r(s') > 0$ holds. The facts that $transAtPr$ can execute γ and that $r(s') > 0$ guarantee that $value(r, \sigma, s) > 0$, because $value$ stops executing σ when no more reward improvement is possible and with s' there is at least one successor situation with positive reward. Consequently, $transPr$ chooses a decomposition that is transitionable at least once and returns its probability. Since the value is positive and the value includes the probability as a factor, this probability must be positive.[3]

The $\supset$-direction (5.14): Assume $transPr(r, \sigma, s, \delta, s') > 0$ holds. Then there is some partition $(\gamma; \delta)$ of σ such that γ can be executed by $transAtPr$ in s leading to s'. This means

$$Next(\sigma, \gamma, \delta) \wedge transAtPr(r, \gamma, \delta, s, s') > 0$$

holds. With Lemma 5.7 the claim follows. □

Corollary 5.10. *$Trans^*$ can execute a non-stochastic program σ if $transPr^*$ can:*

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models \exists \delta, s' \,.\, transPr^*(r, \sigma, s, \delta, s') > 0 \supset Trans^*(\sigma, s, \delta, s').$$

Proof. Theorem 5.9 propagates to the recursive closures of $transPr$ and $Trans$. □

Corollary 5.11. *If $doPr$ can execute a program σ, Do can execute σ, too, as long as it is non-stochastic:*

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{C}' \models \exists \delta, s' \,.\, doPr(r, \sigma, s, \delta, s') > 0 \supset Do(\sigma, s, s').$$

[3] If $r(s') = 0$, $transPr$ might encounter a scenario in which all alternatives give a reward of 0. For example, $transPr$ might branch to σ_1 in $\sigma_1 \,|\, \sigma_2$ because

$$value(r, \sigma_1, s) = 0 \geq 0 = value(r, \sigma_2, s).$$

However, it might be the case that σ_1 could not be transitioned even once, whereas σ_2 might fail later. In this scenario, $Trans$ would perform a single transition of the program by branching to σ_2, while $transPr$ would fail immediately, that is, return 0.

Proof. Both, *doPr* and *Do*, use the transitive closures of $transPr$ and $Trans$, respectively. *doPr*, however, may fail earlier than *Do*, namely when the program is final with respect to the $transPr$ semantics. This is the case when executing the program does not improve the estimated reward (cf. $Final$ (5.12)). □

Note that the reverse does not hold, that is, executability with respect to *Do* does not imply executability with respect to *doPr*. This is because the reward function r could *doPr* "trap" the interpreter and lead to a non-executable action.

Short Summary

The presented semantics of stochastic actions has the following distinguishing features:

- The probability distribution refers to the non-time-stamped actions. For this reason, τ is chosen deterministically in the α transition.
- Nondeterminism at agent's choice points is resolved by choosing the branch that maximizes the reward very similar to DTGolog.
- In contrast to DTGolog, concurrency is supported. The $Next$ predicate extracts all possible next atomic actions (stochastic, primitive or test) and their remainders, so that concurrency is pushed back to the remainder continuously. For example, the nondeterministic program $\sigma_1 \parallel \sigma_2$ could be converted into $(\gamma_1; (\delta_1 \parallel \sigma_2))$ or $(\gamma_2; (\sigma_1 \parallel \delta_2))$ where the next transition of γ_i is deterministic.

5.4.4 Atomic Complex Actions

In general imperative programming, complex operations can be made atomic using, for example, semaphores or test-and-set instructions. Many modern programming languages provide higher-level constructs for mutual exclusion such as Ada 95's protected types (Taft and Duff, 1997, p. 159) and Java's `synchronized` keyword (Gosling et al., 2005, p. 554). The following presents a high-level Golog construct concerning this matter.

The above semantics supports concurrency. Critical sections can be guarded by special purpose fluents that represent locks. Let there be two actions, *lock* and *unlock*, and one fluent *Locked* with obvious meanings. The lock is then implemented in *lock*'s precondition and *Locked*'s successor state axiom:

$$\begin{aligned}
Poss(lock, s) &\equiv \neg Locked(s) \\
Poss(unlock, s) &\equiv True \\
Locked(do(a, s)) &\equiv a = lock \vee Locked(s) \wedge a \neq unlock.
\end{aligned}$$

This appears to be similarly tedious like low-level locking in traditional programming languages. Therefore, it would be desirable to define a new keyword that marks sections that are executed atomically.

The semantics defined in Section 5.4.3 is based on the decomposition of a program σ into a next atomic action γ and a remaining program δ. This is done by $Next$. Up to now, only primitive, stochastic and test actions were considered atomic. From now on, programs of the form $atomic(\sigma)$ are meant by atomic, too. Hence, $Next$'s definition (5.7) can be extended with

$$Next(atomic(\sigma), \gamma, \delta) \equiv \gamma = atomic(\sigma) \land \delta = Nil.$$

$transAtPr$ can be modified to handle the new atomic actions $atomic(\sigma)$:

$$transAtPr(r, atomic(\sigma), \delta, s, s') = p \equiv transPr^*(r, \sigma; \delta, s, \delta, s') = p.$$

Handing over δ to $transPr^*$ as parameter for the remaining program ensures that only the complex action σ is executed and no part of δ. This modification, however, means that neither $transAtPr$ nor $transPr$ necessarily perform a exactly one transition.

Hence, looking for an alternative way of implementing the *atomic* construct is worthwhile. Instead of adjusting $transAtPr$, one can replace the $Next$ used in $transPr$'s definition (5.11) with $Next'$ defined as

$$\begin{aligned} Next'(\sigma, \gamma, \delta) \overset{\text{def}}{=} \forall P\,.\, & \big(\forall \sigma', \gamma', \delta'\,.\, Next(\sigma', \gamma', \delta') \supset P(\sigma', \gamma', \delta')\big) \land \\ & \big(\forall \sigma', \sigma'', \gamma', \gamma'', \delta', \delta''\,. \\ & \quad P(\sigma', \gamma', \delta') \land \gamma' = atomic(\sigma'') \land Next(\sigma''; \delta', \gamma'', \delta'') \supset \\ & \quad\quad P(\sigma', \gamma'', \delta'')\big) \supset \\ & P(\sigma, \gamma, \delta) \land (\forall \sigma')\gamma \neq atomic(\sigma'). \end{aligned} \tag{5.15}$$

P denotes the transitive closure of re-applying $Next$ if γ' is a complex atomic action. For each complex atomic action $\gamma' = atomic(\sigma'')$, $Next$ is recursively applied to $\sigma''; \delta'$.

Even if σ' contains concurrency above the level of $\gamma' = atomic(\sigma'')$, then γ' and thus σ'' are free of this concurrency, because $Next$ pushes back the concurrency to δ'. The subsequent call to $Next$ for $\sigma''; \delta'$ does not change this, because concurrency in δ' cannot spread to σ''.

Since $Next'$ is only called from $transPr$, it is guaranteed that there is no other program running concurrently with σ at the time of calling.

5.5 Summary

This chapter proposed a semantics that enriches ConGolog with a flexible, descriptive, continuous model of time (cf. Section 5.1). After showing that this notion of time is not contrary to nondeterminism and concurrency (cf. Section 5.2, Section 5.3), DTGolog-like stochastic actions and decision theory were integrated (cf. Section 5.4).

Combining DTGolog with concurrency is mentioned as future work in (Boutilier et al., 2000a,b). It turned out that it is necessary to abandon the $Trans$ semantics in favor of a new $Next$ predicate for program decomposition, $transAtPr$ for execution of primitive, stochastic and test actions, and a relatively concise definition of $transPr$ that puts things together (cf. Section 5.4.3). A lucky side effect of this semantics is that complex atomic actions can be supported by the language as described in Section 5.4.4.

The final version of the semantics is given in Section 5.4.3 with a little modification in Section 5.4.4.

The presented action language is influenced by several existing Golog dialects: ConGolog for concurrency (Giacomo et al., 2000), temporal sequential Golog and ccGolog for time (Reiter, 1998; Grosskreutz and Lakemeyer, 2000a, 2003), and finally stGolog, DTGolog and pGolog for robustness (Reiter, 2001; Boutilier et al., 2000a,b; Grosskreutz, 2000; Grosskreutz and Lakemeyer, 2000b).

Chapter 6

Plan Recognition

Chapter 5 defined the semantics of a Golog dialect that handles programs like the passing maneuver from Figure 4.2 on page 26 and defines some features beyond. Up to now, this was not specific to plan recognition, although the semantics was designed with plan recognition in mind.

This chapter presents a new approach to plan recognition based on this language: a program is a solution of the plan recognition problem if its execution is consistent with the observations.

The first section gives an intuitive definition of plan recognition in the form a first-order logic formula. Subsequent ways of plan recognition are compared with this formula's semantics to verify their reasonableness.

The second section develops a procedure for incremental plan recognition based on iteratively filtering a set of candidates in the fashion of Goultiaeva and Lespérance (2006). Up to this point, the non-probabilistic semantics is used, that is, the $Trans$ predicate before Section 5.4.

Section 6.3 then integrates plan recognition into the Golog program. This is done by exploiting the decision-theoretic properties of the new $transPr$ semantics from Section 5.4.3. All following parts rely on the *plan recognition by program execution* approach developed in this section. In particular, the simple heuristic defined in this section is used in our implementation described in the next chapter.

After the next section shortly describes how multiple agents are modeled in this framework, Section 6.5 uses the probabilistic features of the $transPr$ semantics to do robust plan recognition. For one, this allows to handle discrepancies between the agent's actions and the program, and for another it provides a way to model sensor noise.

Finally, an example of how the plan recognition system works in a very simple world is given in Section 6.6.

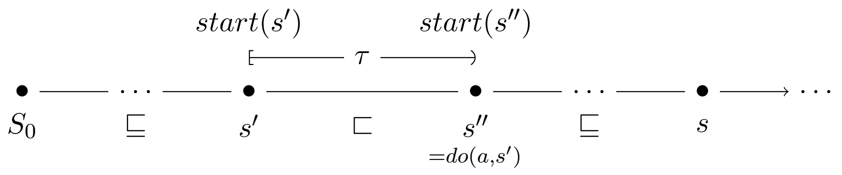


Figure 6.1: $TimeSit(s', \tau, s)$ holds if τ belongs to s' on the way to s.

6.1 Plan Recognition by Satisfiability

The plan recognition presented by Goultiaeva and Lespérance (2006) expects an observed situation as input which is basically a sequence of primitive actions. They then iteratively filter the set of potentially executed plans by checking whether or not the next observed action might be the next action of the respective program.

However, directly observing actions is not realistic in automobile scenarios, although in the future such information might be distributed via car-to-car communication. At least today, periodic observations of the environment seem to be more realistic. An observation at time T is a first-order sentence Φ that describes what is true in the real world at point in time T. The fluents in Φ are all situation- and time-suppressed.

Observation sentences may be of any form, particularly, they may be disjunctions. This is simply due to the fact that observations are only tested, they are not added to the knowledge base. In reality, though, observations are rather conjunctions such as $pos = (50, -3) \land veloc = 25 \land yaw = 0$.

The following macro asserts that time τ belongs to situation s' on the way to situation s as visualized in Figure 6.1:

$$TimeSit(s', \tau, s) \overset{\mathsf{def}}{=} s' \sqsubseteq s \land \tau \geq start(s') \land \big(\forall s'' \,.\, s' \sqsubset s'' \land s'' \sqsubseteq s \supset \tau < start(s'')\big). \tag{6.1}$$

The problem of plan recognition then reduces to finding a program σ such that

$$\mathcal{D} \cup \mathcal{C} \models \exists s \,.\, Do(\sigma, S_0, s) \land \exists s' \,.\, TimeSit(s', \mathrm{T}, s) \land \Phi[s', \mathrm{T}]$$

holds. T is not quantified because it is the observation's timestamp. If there is such a program σ, then during its execution there will be a situation at time T in which Φ holds, i.e. the situation matches the observation at time T.

If there are multiple observations $\Phi_1, \ldots, \Phi_n$, then the task is to find a program σ such that

$$\mathcal{D} \cup \mathcal{C} \models \exists s \,.\, Do(\sigma, S_0, s) \land \exists s_1, \ldots, s_n \,.\, \bigwedge_{i=1}^{n} TimeSit(s_i, \mathrm{T}_i, s) \land \Phi_i[s_i, \mathrm{T}_i] \tag{6.2}$$

holds.

6.2 Incremental Plan Recognition

Formula (6.2) does not provide any means for incremental plan recognition, i.e. for handling a stream of incoming observations. In style of Goultiaeva and Lespérance (2006), we define a set *allConsistPlans* which contains all situations consistent with the observations and the accompanying remaining program of the plan that resulted in the respective situation. As mentioned above, the approach of Goultiaeva and Lespérance (2006) expects a stream of actions as input. Since here observations are logical formulas with a corresponding timestamp, the input of *allConsistPlans* is a stream of observations.

If there is a unique model M_0 such that $M_0 \models \mathcal{D} \cup \mathcal{C}$, i.e. if information about the initial situation S_0 is complete, *allConsistPlans* can be defined as follows:

$$\begin{aligned}
&\mathit{allConsistPlans}([\,]) = \{(\sigma, S_0) \mid \sigma \in \text{plan library}\} \qquad (6.3)\\
&\mathit{allConsistPlans}([(\Phi,\mathrm{T}) \mid \Psi]) = \{(\delta, S') \mid (\sigma, S) \in \mathit{allConsistPlans}(\Psi) \text{ and}\\
&\qquad M_0 \models \exists s''.\, \mathit{Trans}^*(\sigma, S, \delta, s'') \wedge\\
&\qquad\qquad \mathit{Trans}^*(\mathit{waitFor}(\mathit{now} = \mathrm{T}); \Phi?, s'', \mathit{Nil}, S')\}.
\end{aligned}$$

In this definition, $[\,]$ denotes the empty list and $[(\Phi,\mathrm{T}) \mid \Psi]$ takes the head element, the observation of formula Φ at point in time T, out of the list such that Ψ is the rest. This list is assumed to be ordered latest-observation-first. $\mathcal{C}$ and *Trans* are those defined before Section 5.4, that is, the normal transitional semantics with time, nondeterminism and concurrency (cf. Section 5.1, Section 5.2, Section 5.3). Furthermore, the fluent *now* is the identity of the time: $\mathit{now}(s) = \mathit{linear}(0,1,0)$. Recall that according to the definition of *linear* in Section 4.1, *now* is evaluated in a situation s at time τ the following way:

$$\mathit{val}(\mathit{now}(s), \tau) = \mathit{val}(\mathit{linear}(0,1,0), \tau) = 0 + 1 \cdot (\tau - 0) = \tau.$$

It is not sufficient to define $\mathit{now}(s) = \mathit{start}(s)$.

Strictly speaking, the test of Φ? does not work, because Φ involves continuous fluents which need to be evaluated. There is a number of solutions to this small problem. One way would be to modify the *Trans* clause for tests (3.1) so that continuous fluents are evaluated. Replacing $\phi[s]$ with $\phi[s, \mathit{start}(s)]$ is enough for that. An alternative way is to move the test simply into the *waitFor* action, thus getting $\mathit{waitFor}(\mathit{now} = \mathrm{T} \wedge \Phi)$. The definition of *allConsistPlans* refrains from this solution in favor of the "illegal" test of Φ in order to make clear that the interpreter does not *wait* for Φ getting true.

The above formula takes a remaining program σ and the corresponding situation S from a recursive call and runs σ until the latest observation Φ at time T holds. This is done as follows: The first call to $Trans^*$ determines the remainder of σ and an intermediate situation s''. Starting from s'', the $waitFor(now = \mathrm{T})$ shifts time to the point of the latest observation and the subsequent test $\Phi?$ asserts that the observation holds in S' at time T. The $waitFor$ is necessary to enforce future executions starting at S' to begin not before time T.

This definition is not equivalent to the definitions from Section 6.1 due to a small difference: According to the definition of the $TimeSit$ (6.1), the time interval of a situation excludes the start of the successive situation. From this follows that for two situations $s \sqsubset s'$ with $start(s) = start(s')$, an observation is never tested in s but only in s'. The idea is that the actions that lead from s to s' might influence the truth value of the observation. For example, a database might be inconsistent in s but consistency could be re-established immediately in s'. An observation of an inconsistent database should therefore not be entailed, because there was no point in time at which this could have been observed. Using $allConsistPlans$, however, this is not the case, because inconsistency holds in s.

A way to simulate the behavior of $TimeSit$ (6.1) is to additionally execute a $waitFor$ action that enforces time to advance really after T. This can be incorporated into Formula (6.3) as follows:

$$\begin{aligned}&allConsistPlans([(\Phi, \mathrm{T}) \mid \Psi]) = \{(\delta, S') \mid (\sigma, S) \in allConsistPlans(\Psi) \text{ and} \\ &\quad M_0 \models \exists s'' . Trans^*(\sigma, S, \delta, s'') \wedge \\ &\qquad\qquad Trans^*(waitFor(now = \mathrm{T}); \Phi?; waitFor(now > \mathrm{T}), s'', Nil, S')\}.\end{aligned} \tag{6.4}$$

Modifying the conjunction modeled by M_0 in Formula (6.3), for example by adding

$$\forall s''' . Trans^*(\delta, s'', \delta', s''') \supset \big(\exists a . s''' = do(a, s'') \supset start(s'') < start(s''')\big),$$

does not do the trick. The problem is that at this state, it is not yet known which branch of δ will be taken. If $\delta = \alpha_1; \alpha_2^*$ and α_2 is executable immediately after α_1, the above formula will fail. One could greedily apply $Trans$ until no subsequent immediate action is possible, but this would constrain the time variables of "future" actions too early.

6.3 Plan Recognition by Program Execution

The concept of $allConsistPlans$ can be merged into the program itself. At first, non-incremental plan recognition based on executing a specific program is described. The next subsection provides a way for incremental plan recognition exploiting the decision-theoretic part of the semantics from Section 5.4.3. Finally, the last subsection suggests how heuristic and thus realistic plan recognition can be carried out.

While to the previous two sections used the *Trans* semantics, this section starts using the *transPr* semantics due to its decision-theoretic resolving of nondeterminism and atomic actions.

6.3.1 Non-Incremental Plan Recognition by Program Execution

Observations can actually be converted into a program itself. Each observation ϕ at time τ is represented by an action $\mathit{observe}(\tau, \phi)$ whose precondition is

$$\mathit{Poss}(\mathit{observe}(\tau, \phi, \tau'), s) \equiv \tau = \tau' \wedge \phi[s, \tau].$$

The *observe* action is very similar to the aforementioned *waitFor* action. Both have no explicit effects in terms of effect axioms, but they force time to advance to a certain point in time τ. But for one, the name *observe* better captures the intention of such an action, and for other, the action name will be used in the incremental plan recognition to distinguish it from other actions.

A sequence of observations $\phi_1, \ldots, \phi_n$ with accompanying timestamps $\tau_1, \ldots, \tau_n$ is turned into a program

$$\theta = \mathit{observe}(\tau_1, \phi_1); \ldots; \mathit{observe}(\tau_n, \phi_n).$$

A program σ is consistent with these observations, if $\sigma \parallel \theta$ can be executed by the interpreter.

The action $\mathit{observe}(\mathrm{T}, \Phi)$ very much resembles the combination of $\mathit{waitFor}(\mathit{now} = \mathrm{T})$ and Φ? in *allConsistPlans*. In Formula (6.3), it is valid to split $\mathit{now} = \mathrm{T}$ and Φ into two actions, because they are executed isolated from the main program σ. In $\sigma \parallel \theta$, however, atomicity of observation-specific actions needs to be ensured to prevent that an action of σ could screw itself in between a *waitFor* and a test action. The primitive action *observe* is atomic by definition.

The semantics of $\sigma \parallel \theta$ correspond to the original definition of *allConsistPlans* in (6.3). To correctly emulate *TimeSit*'s (6.1) behavior, (6.3) is slightly changed to (6.4) by adding an action $\mathit{waitFor}(\mathit{now} > \mathrm{T})$ which prevents an observation from being entailed by a situation with a timespan of zero. Using the new atomic complex actions from Section 5.4.4, the required atomicity can be ensured:

$$\mathit{atomic}(\mathit{observe}(\mathrm{T}, \Phi); \mathit{waitFor}(\mathit{now} > \mathrm{T})).$$

This avoids that any other action from the candidate program σ gets between $\mathit{observe}(\mathrm{T}, \Phi)$ and $\mathit{waitFor}(\mathit{now} > \mathrm{T})$.

6.3.2 Incremental Plan Recognition by Incremental Program Execution

As argued in Section 6.2, observations are not known in advance but come in as a stream. Hence, the observation program cannot be constructed in advance either. To solve this problem, the candidate program σ, whose consistency with the observations is to be checked, could be executed incrementally using $\mathit{transPr}^*$. Each time new observations occur, these need to be merged into the program.

Given a candidate program σ from the plan library, the initial state of the plan recognition system is $\{(\sigma, S_0, 1)\}$, meaning that the program σ needs to be executed starting in S_0 and the probability of σ being successfully executed up to now is 1. The recursion proceeds as follows: If (δ, s, p) is in the current state and the observation ϕ with τ just came in, (δ', s', p') is added to the successor state for all

$$p' = p \cdot \mathit{transPr}^*(r, \delta \circ \mathit{observe}(\tau, \phi), s, \delta', s')$$

where $\circ$ is an operation defined below that merges δ and the observation. Note that since $\mathit{transPr}^*$ also succeeds for zero applications of $\mathit{transPr}$, each successor state also contains its predecessor.

Without loss of generality, we assume that no two observations have the same timestamps. Observations with equal timestamps can simply be merged into a single *observe* action by conjunction.

The standard reward function (5.3) is not applicable in this scenario where observations are not known in advance. A suitable reward function is

$$r(s) = \begin{cases} 0 & \text{if } s = S_0 \\ 1 + r(s') & \text{else if } (\exists \tau, \phi, s') s = \mathit{do}(\mathit{observe}(\tau, \phi, \tau), s') \\ r(s') & \text{else.} \end{cases} \tag{6.5}$$

which counts the number of *observe* actions in the situation term s. This reward is the number of entailed observations. Obviously, it can be easily implemented as functional fluent.

Before *allConsistPlans* is defined for incremental program execution, the merge operation $\circ$ is still left to be defined. The solution is to execute σ and the *observe* action concurrently:

$$p' = p \cdot \mathit{transPr}^*(r, \delta \parallel \mathit{observe}(\tau, \phi), s, \delta', s').$$

The following results show that the sequential observation program from Section 6.3.1 and merging the *observe* actions into the program using the concurrency operator are equivalent.

Lemma 6.1. *Concurrency is an commutative and associative operation:*

$$\mathcal{D} \cup \mathcal{C} \models \mathit{transPr}(r, \sigma_1 \parallel \sigma_2, s, \delta, s') = \mathit{transPr}(r, \sigma_2 \parallel \sigma_1, s, \delta, s').$$

and

$$\mathcal{D} \cup \mathcal{C} \models \mathit{transPr}(r, \sigma_1 \parallel (\sigma_2 \parallel \sigma_3), s, \delta, s') = \mathit{transPr}(r, (\sigma_1 \parallel \sigma_2) \parallel \sigma_3, s, \delta, s').$$

Proof. Both properties follow from the definition of Next. □

Lemma 6.2. *Given n subsequent observation actions $O_1, \dots, O_n$ with timestamps $\tau_1 < \dots < \tau_n$, sequential and concurrent execution behave the same way:*

$$\mathcal{D} \cup \mathcal{C} \models \forall \sigma \,.\, \mathit{doPr}(r, \sigma \parallel (O_1; \dots ; O_n), s, \delta, s') = \\ \mathit{doPr}(r, \sigma \parallel (O_1 \parallel \dots \parallel O_n), s, \delta, s').$$

Proof. If executing $\sigma \parallel (O_1; \dots ; O_n)$ returns 0, then $\sigma \parallel (O_1 \parallel \dots \parallel O_n)$ returns 0, too.

If executing $\sigma \parallel (O_1 \parallel \dots \parallel O_n)$ returns $p > 0$, particularly $O_1, \dots, O_n$ succeed. Since *observe*'s precondition requires each O_i to be executed at time τ_i, the only possible ordering is $O_1; \dots ; O_n$. Therefore, the same probability $p > 0$ is returned by the sequential program. □

Theorem 6.3. *For a candidate program σ and observation actions $O_1, \dots, O_n$, sequential and concurrent execution of the O_i are equivalent:*

$$\mathcal{D} \cup \mathcal{C} \models \mathit{doPr}(r, \sigma \parallel (O_1; \dots ; O_n), s, \delta, s') = \\ \mathit{doPr}(r, ((\sigma \parallel O_{i_1}) \parallel \dots \parallel O_{i_n}), s, \delta, s')$$

where $\{i_1, \dots, i_n\} = \{1, \dots, n\}$.

Proof. Due to Lemma 6.1, the program $(((\sigma \parallel O_1) \parallel O_2) \dots \parallel O_n)$ can be brought into the form $\sigma \parallel (O_1 \parallel \dots \parallel (O_{n-1} \parallel O_n))$. According to Lemma 6.2, the concurrency operators connecting the O_{i_j} can be replaced with sequence operators. □

Finally, the incrementally growing set of consistent plans *allConsistPlans* originally defined in Formula (6.3) can be redefined as follows:

$$\begin{aligned} &\mathit{allConsistPlans}([\,]) = \{(\sigma, S_0, 1) \mid \sigma \in \text{plan library}\} \\ &\mathit{allConsistPlans}([(\Phi, \mathrm{T}) \mid \Psi]) = \{(\delta, S', P \cdot P') \mid \\ &\quad (\sigma, S, P) \in \mathit{allConsistPlans}(\Psi) \text{ and} \\ &\quad M_0 \models \mathit{transPr}^*(r, \sigma \parallel \mathit{observe}(\mathrm{T}, \Phi), S, \delta, S') = P' \wedge P' > 0\}. \end{aligned}$$

Again, to make it completely compatible with *TimeSit* (6.1), the *atomic* construct can be used:

$$\begin{aligned}
&allConsistPlans([(\Phi, \mathrm{T}) \mid \Psi]) = \{(\delta, S', P \cdot P') \mid \\
&\quad (\sigma, S, P) \in allConsistPlans(\Psi) \text{ and} \\
&\quad M_0 \models transPr^*(r, \sigma \parallel atomic(observe(\mathrm{T}, \Phi); waitFor(now > \mathrm{T})), \\
&\qquad\qquad\qquad S, \delta, S') = P' \wedge P' > 0\}.
\end{aligned}$$

6.3.3 Heuristic Incremental Plan Recognition

In practice, the set *allConsistPlans* tends to become too large. This is because it never commits itself to some state. Instead, there are two sources of the blowup:

- Stochastic actions may have countably infinitely many outcome actions. For each outcome action, there is a single distinguished successor configuration in *allConsistPlans*.
- *transPr** performs a nondeterministic number of transitions. Therefore, for each possible number of transitions, (at least) one configuration occurs in *allConsist-Plans*. Particularly early configurations such as $(\sigma, S_0, 1)$ always remain in the set.

With respect to the first issue, a real world implementation could either restrict the axiomatizer to a subclass of stochastic actions so that it suffices to always consider one outcome action, for example the most probable one. However, it appears difficult to find an expressive class due to the interdependencies of effects and preconditions. An alternative solution is sampling. Program execution is simply repeated a number of times, and for each encountered stochastic action, an outcome action is picked at random. The following assumes that, whatever solution is chosen, a stochastic action leads only to a single successor configuration just like normal primitive actions.

Regarding the second point, it is clearly not practical that each configuration is kept in the *allConsistPlans* set. Instead of using *transPr**, one might apply *transPr* whenever the situation allows it.

At this point, reward values and probabilities of *transPr* from Section 5.4.3 come in very handy. We assume that the reward function (6.5) is used which counts the number of observations explained so far.

First of all, a horizon should be introduced for the *value* function. As a consequence, the interpreter might not choose the optimal branch in all cases, but optimizing the value with respect to the complete program is not only computationally expensive, but even impossible when observations are not known in advance.

Then, if h is the horizon, h observations should be kept in a queue during execution at any time. Observations must be held back so that the interpreter can do a reasonable job resolving nondeterminism. This is because the reward function (6.5) returns the number of *observe* actions in the given situation term, and therefore *value* determines the estimated reward of a program with a lookahead of h transitions.

Hence, instead of using $transPr^*$ with its whole string of successor configurations, the interpreter may always trigger a transition whenever at least h observations are in the queue. After each transition, the interpreter might need to wait until the queue of observations has grown to at least h elements and may then proceed with the next transition. This for one guarantees that the interpreter always does an informed choice, and for other it rules out the possibly vast number of successor triples that are due to succeeding and preceding situations returned by $transPr^*$.

Due to this heuristic, the state of the plan recognition system no longer is the evergrowing *allConsistPlans*. Instead, for each candidate program σ, there is only one triple (δ, s, p) in each state (ignoring stochastic actions for the aforementioned reasons).

Since committing to a unique configuration runs the danger of leading to a bad choice, the interpreter may keep the n best configurations per program at any point of time. Note that "best" is meant with respect to the estimated reward, which is exactly the criterion by which *transPr* resolves nondeterminism. We assume that at each iteration, a list of n configurations exists that represents the best ones up to now. Hence, we start off with n copies of the initial configuration $(\sigma, S_0, 1)$. Whenever enough observations are buffered so that a transition can be triggered, for each configuration the n best successors are determined. For this, *transPr* needs be modified appropriately so that it not only computes the single best successor configuration but the n best configurations. Out of these $n \cdot n$ new configurations, we pick the n best ones, that is, those triplets (σ, s, p) who maximize the estimated reward $p \cdot value(r, \sigma, s)$. Then, the system waits for the next observation and repeats this procedure.

The rest of this subsection is a runtime analysis depending on the horizon.

For each function *transPr*, *transAtPr*, *value* and $transPr^*$ we add a new version with superscript h which denotes the current horizon. Hence, $transPr^{*h}$ is a variant of $transPr^*$ which performs a nondeterministic number of 0 to h transitions. This function is called in $value^h$, which again is called by $transPr^{h+1}$ and $transAtPr^h$. This implements the limited lookahead.

A real world implementation of $transPr^h$ might work like this: After determining all decompositions $(\gamma; \delta)$ of the input program, the (non-complex) atomic action γ is executed using $transAtPr^{h-1}$, and then $value^{h-1}$ is computed for the remaining program δ, which by means of $transPr^{*h-1}$ leads to a recursive call of $transPr^{h-1}$.

Ignoring the call to $value^{h-1}$ in $transAtPr^{h-1}$'s rule for primitive actions (5.8),[1] the cost of executing γ is $\mathcal{O}(1)$. However, $transPr^h$ still needs to call $value^{h-1}$ which calls

[1]This is quite realistic, because timestamps are solver variables in the implementation and not directly subject to value optimization.

$\mathit{transPr^*}^{h-1}$ and ultimately leads to $\mathit{transPr}^{h-1}$. The number of transitions induced by $\mathit{transPr}^h$ is thus $\mathcal{O}(t_1(h))$ for

$$t_1(h) = K \cdot (1 + t_2(h-1))$$

where $t_2(h)$ is the number of transitions induces by $\mathit{transPr^*}^h$:

$$t_2(h) = \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } t_1(h) + t_2(h).$$

The equations can be resolved starting with $t_2(h)$:

$$\begin{aligned}
t_2(h) &= \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } t_1(h) + t_2(h-1) \\
&= \sum_{i=1}^{n} t_1(i) \\
&= \sum_{i=1}^{n} K \cdot (1 + t_2(i-1)) \\
&= h \cdot K + K \cdot \sum_{i=1}^{n} t_2(h-1) \\
&= h \cdot K + K \cdot \big[(h-1) \cdot K + K \cdot \big[(h-2) \cdot K + K \cdot \big[\ldots \cdot \big[K + K \cdot t(0)\big] \ldots\big]\big]\big] \\
&= h \cdot K^1 + (h-1) \cdot K^2 + (h-2) \cdot K^3 + \ldots + K^h + 0 \\
&= \sum_{i=1}^{h} (h - i + 1) \cdot K^i \\
&\leq h \cdot K^{h+1}.
\end{aligned}$$

Therefore, $t_2(h) = \mathcal{O}(h \cdot K^{h+1})$ and $t_1(h) = \mathcal{O}(K + (h-1) \cdot K^{h+1}) = \mathcal{O}(h \cdot K^h)$. This means that the computational cost of optimization in a single transition step increases by $\mathcal{O}(h \cdot K^h)$ with the horizon h.

6.4 Multi-Agent Plan Recognition

In many scenarios such as automotive highway traffic, multiple drivers act at once.

Observations can easily refer to multiple different drivers since they simply are situation- and time-suppressed logical formulas.

To extend the plan recognition framework with support for multi-agent scenarios, one can exploit the concurrency operator. If there are programs $\sigma_1(v), \ldots, \sigma_n(v)$ for each agent v, the plan recognizing Golog interpreter can be called for the program $\sigma_1(v) \,|\, \ldots \,|\, \sigma_n(v)$. The interpreter branches to that σ_i that explains the observations at best. If there are now multiple agents $v_1, \ldots, v_m$ active at once, all agents' programs simply need to be executed concurrently:

$$\big(\sigma_1(v_1) \,|\, \ldots \,|\, \sigma_n(v_1)\big) \parallel \ldots \parallel \big(\sigma_1(v_m) \,|\, \ldots \,|\, \sigma_n(v_m)\big).$$

This also captures interdependencies of the different agents' actions.

6.5 Data Robustness

The probabilistic extension of the Golog interpreter described in Section 5.4.3 is motivated by the need of robustness towards data errors. This section exemplarily shows how a certain permissiveness towards the agent's actions can be modeled using the probabilistic semantics. Additionally, it is shown how sensor error profiles can be modeled. Note that the following subsections are not mutually exclusive but complement each other.

6.5.1 Tolerance towards Agent's Fuzziness

While it appears to be intuitive to describe a passing maneuver as simple and definitive as in Figure 4.1 on page 23, a real driver cannot be expected to drive exactly this way. Even in simpler scenarios, reality differs from an idealized description: When a car drives straight ahead, the corresponding program might be as simple as $setYaw(0°)$, which means that the vehicle's yaw is exactly 0°. This is exactly the driver's actual *goal*. In reality, however, the yaw differs at least slightly from 0°. Besides human inaccuracy, real world influences such as wind and bumps can be responsible for these deviations. The driver controls his car so that the yaw *averages* 0°. In a nutshell, the real world (the driver) is less perfect than the model (the program).

In a sense, the reverse sentence unfortunately holds, too: The model is less perfect than reality. In theory, it is easy to develop a physical vehicle model for an idealized world without wind and other external influences. In reality, however, this quickly involves nonlinear (in)equations. For example, given a uniform acceleration, the distance depends on the time squared. Sticking with linear programs appears to be worthwhile because they can be solved efficiently (Zimmermann, 2005).

Trying to bring model and real world in line and basing plan recognition alone on the question whether or not this is possible appears to be insufficient. The degree by which the model and the real world comply should be measured.

The probabilities of Section 5.4.3 allow to do this. Using *plan recognition by program execution* as described in Section 6.3, the degree by which a given program explains the observations should be simply the probability by which the program can be executed.

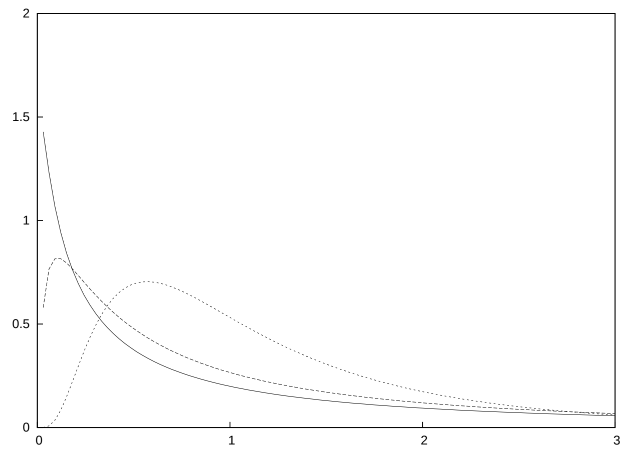
The general idea is to introduce randomly distributed tolerances. The *ovserve* action's precondition needs to be altered to consider the tolerances: $\mathit{observe}(\tau, \phi, \tau')$ is executable if ϕ' holds for ϕ' being the variant of ϕ that incorporates the tolerances. Usually, high tolerances are intended to be less likely than small tolerances. If an observation holds for a small tolerance, it also holds for a high tolerance. Consequently, if the observations match exactly the model, the probability returned by the interpreter is 100 %.

One way to integrate tolerances is using intervals instead of atomic numbers. The fluent function $y(s)$ then would be split into a lower bound $y^-(s)$ and an upper bound $y^+(s)$. The successor state axioms then use interval arithmetic instead of normal arithmetic. For example, a formula $y(s) = Y$ is translated to $y^-(s) \leq Y \leq y^+(s)$. The tolerances come into the fluent values by stochastic actions. The aforementioned $\mathit{setYaw}(\gamma)$ action then becomes a stochastic action whose outcomes are $\mathit{setYaw}(\gamma - t, \gamma + t)$ where t is the tolerance. A sample probability distribution for t is depicted in Figure 6.2. The term $\gamma - t$ denotes the lower bound of the vehicle's yaw, $\gamma + t$ is the upper bound. Considering the above example of the straightforward driving vehicle, these intervals model the driver's deviations from 0°. However, they do not capture that the driver controls his car to *average* to 0°. Instead, interval arithmetic multiplicates the tolerance. A very simple example illustrates this problem: if the current situation s with $y^-(s) = y^+(s) = 0\,\mathrm{m}$ and the outcome action of $\mathit{setYaw}(0°)$ is $\mathit{setYaw}(-1°, 1°)$, then after 100 m, the bounds have already diverged by about 3.5 m which is pretty much with respect to a typical lane's width.

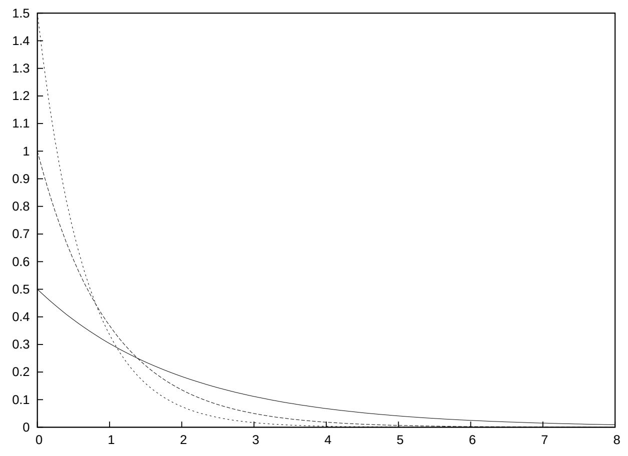
An alternative approach is to stick with the model's trajectory of the car and add a tolerance area around the vehicle. For simplicity, this area is assumed to be a rectangle specified by the tolerance in X- and Y-direction. Actions have effects on the size of this area. For example, during a lane change the rectangle should grow along the Y-axis. This behavior is depicted in Figure 6.3. The increase of the Y-coordinate is needed, because the real vehicle smoothly starts and ends each lane change, while the model vehicle changes its direction instantaneously.

This notion of tolerance area can be integrated into the basic action theory described in Section 4.1 by adding a two fluents, pos^- and pos^+ which return the bottom left and top right points of the tolerance rectangle, respectively. For the sake of brevity, the symbol $\pm$ stands either for $+$ or for $-$ in the following successor state axiom:

$$
\begin{aligned}
(x_1^{t,\pm}, y_1^{t,\pm}) = \mathit{pos}^{\pm}(\mathit{do}(a, s)) \equiv \exists \tau_0, x_0^t, y_0^t, x_0, y_0, v, \gamma, t\,.\\
&\big(a = \mathit{setVeloc}^*(v, t, \tau_0) \land \gamma = \mathit{yaw}(s) \lor \\
&\quad a = \mathit{setYaw}^*(\gamma, t, \tau_0) \land v = \mathit{veloc}(s)\big) \land \\
&(x_0^t, y_0^t) = \mathit{pos}(s) \land \\
&x_0 = \mathit{val}(x_0^t, \tau_0) \land \\
&y_0 = \mathit{val}(y_0^t, \tau_0) \land
\end{aligned}
$$

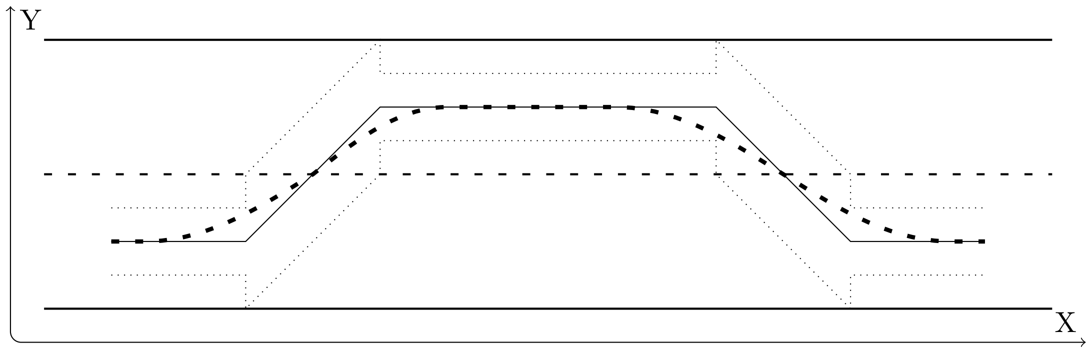


(a) Probability density function of log-normal distribution $\log \mathcal{N}(\mu, \sigma^2)$ for $\mu = 0$ and $\sigma = 2$ (solid), $\sigma = 1.5$ (densely dashed) and $\sigma = 0.75$. On a logarithmic scale, the log-normal distribution looks like as a bell curve. On a linear scale, μ is the location parameter, σ specifies the scale. The mean is $e^{\mu+\sigma^2/2}$, the variance is $(e^{\sigma^2} - 1) \cdot e^{2\mu+\sigma^2}$.

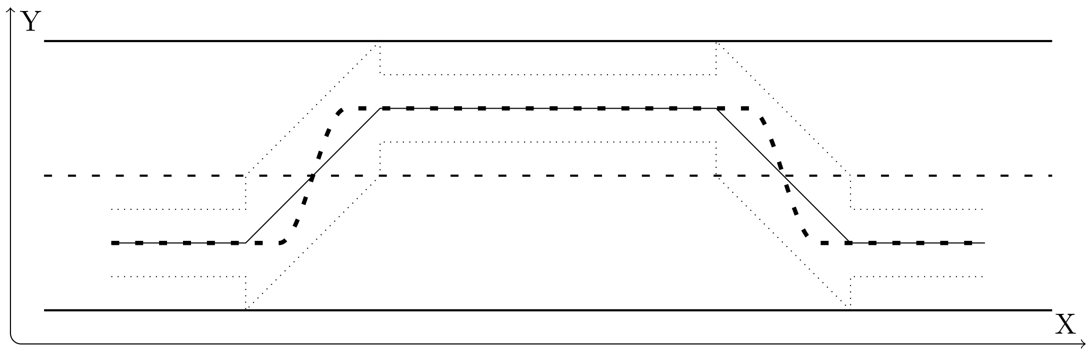


(b) Exponential probability distribution Exp(λ) for $\lambda = 0.5$ (solid), $\lambda = 1$ (densely dashed) and $\lambda = 1.5$. The mean is $\frac{1}{\lambda}$ and the variance is $\frac{1}{\lambda^2}$.

Figure 6.2: Probability density functions of the log-normal and exponential distributions.



(a) Smooth passing maneuver.



(b) Passing maneuver with steep lane changes.

Figure 6.3: Two ways of a real world car (thick dashed lines) to pass another object which is not displayed. The solid line represents the trajectory according to the model. Notice the instantaneous changes of yaw. The lateral tolerance (thin dotted lines) increases during the lane changes, which is needed by the dotted car.

$$\begin{aligned}
x_1^t &= linear(x_0 \pm t, \cos(\gamma) \cdot v, \tau_0) \land \\
y_1^t &= linear(y_0 \pm t, \sin(\gamma) \cdot v, \tau_0) \lor \\
(x_1^{t,\pm}, y_1^{t,\pm}) &= pos(s) \land \\
(\forall v, t, \tau) a &\neq setVeloc^*(v, t, \tau) \land \\
(\forall \gamma, t, \tau) a &\neq setYaw^*(\gamma, t, \tau).
\end{aligned}$$

The binary primitive actions $setVeloc^*$ and $setYaw^*$ are outcome actions of stochastic actions. They replace the old unary primitive actions $setVeloc$ and $setYaw$, respectively, and their additional parameter is ignored in all fluents except for $pos^\pm$. The second parameter, t in the above successor state axiom, denotes the size of the tolerance area. The accompanying stochastic actions are named $setVeloc$ and $setYaw$. The resulting definition of $Choice$ is

$$Choice(setVeloc(v, \lambda), \alpha) \overset{\mathsf{def}}{=} \exists t \,.\, t \geq 0 \land \alpha = setVeloc^*(v, t)$$
$$Choice(setYaw(\gamma, \lambda), \alpha) \overset{\mathsf{def}}{=} \exists t \,.\, t \geq 0 \land \alpha = setYaw^*(\gamma, t).$$

Recall that it is the axiomatizer's job to ensure that the number of outcome actions is at most countably infinite. Therefore, a new sort for tolerance t must be introduced by the axiomatizer, for example the rational numbers. The second parameter of $setVeloc$ and $setYaw$, which is named λ here, is intended to be a parameter for probability distribution of the outcome tolerance t. Thus, the parameter λ of the stochastic action ultimately influences the parameter t of the outcome actions. $prob_0$ could then be

$$prob_0(setVeloc(v, \lambda), \alpha, s) = p \overset{\mathsf{def}}{=} \exists t \,.\, t \geq 0 \land \alpha = setVeloc^*(v, t) \land p = P(\lambda, t)$$
$$prob_0(setYaw(\gamma, \lambda), \alpha, s) = p \overset{\mathsf{def}}{=} \exists t \,.\, t \geq 0 \land \alpha = setYaw^*(\gamma, t) \land p = P(\lambda, t)$$

where P returns a real in the interval $[0, 1]$. For example, $P(\lambda, t)$ might be defined in terms of a random variable X and return $\Pr(X = t)$. The λ can be used to parameterize X's distribution, for example, $X \sim \mathrm{Exp}(\lambda)$ (cf. Figure 6.2b) or, if $\lambda = (\mu, \sigma^2)$, $X \sim \log \mathcal{N}(\mu, \sigma^2)$ (cf. Figure 6.2a). Note that in a program, λ can be instantiated with any value just the same way velocity v and yaw γ are set to some value. In particular, it may increase mean and variance of the probability distribution and therefore effectively enlarge the tolerance area during lane changes.

6.5.2 Sensor Noise with Stochastic Actions

Stochastic actions not only help handling robustness in form of deviations between the program and the actual behavior, they can also be used to model sensors' error profiles.

To achieve this, only the primitive *observe* action needs to be made a stochastic action.

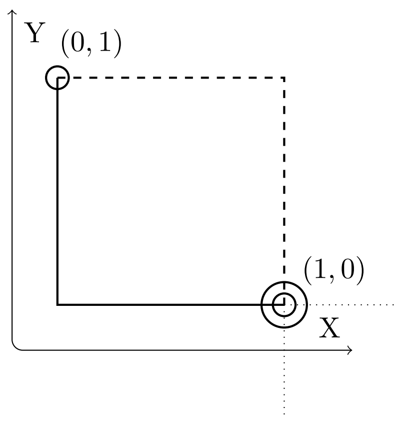


Figure 6.4: Visualization of a very simple continuous world. The agent starts at the circled point and wants to get to the double circled point. It may move either along the straight or the dashed line.

The outcomes of $\mathit{observe}(\tau, \phi)$ then can be primitive actions $\mathit{observe}^*(\tau, \phi^*)$, where ϕ^* is kind of an "outcome formula" of ϕ.

For example, the observation could state that, say, the measured yaw is normally distributed around 10° with standard deviation 2°, that is, $\phi = (\mathit{yaw} \sim \mathcal{N}(10, 2^2)$. Potential outcome formulas then would be $\phi^* = (\mathit{yaw} = 10°)$, $\phi^* = (\mathit{yaw} = 10.5°)$ or $\phi^* = (\mathit{yaw} = -3°)$ with decreasing likelihood. The probability of each ϕ^* is the probability of the respective outcome action $\mathit{observe}^*(\tau, \phi^*)$.

In fact, the Gaussian distribution needs to be discretized in order to comply to the stochastic actions as defined in Section 5.4.3.

6.6 A Simple Example

This example is just a thought experiment intended to illustrate the plan recognition procedure. The world is depicted in Figure 6.4. An agent starts at point at the top left point and wants to get to the bottom right point. To achieve this goal, it can move either right or down with velocity 1 unit/s. Hence, it can move either along the solid line or along the dashed edges. Note that for the sake of simplicity, the agent cannot stop once it has started moving. This particularly means that it is not the goal to *stop* at point $(1, 0)$ but merely to visit this point at some time.

To keep it simple, this example includes neither robustness, particularly no sensor noise and no stochastic actions at all, nor multiple agents. The following is fairly technical, because illustrates the procedure in detail.

6.6.1 The Basic Action Theory

First, we need to develop a basic action theory to formalize the problem. Let there be two actions, R to move right and D to move down. The agent can only move right when it is at the left side of the square. Analogously, it can only move downwards when it is on the top edge. The preconditions of R and D are therefore

$$\begin{aligned} Poss(R(\tau), s) &\equiv val(x(s), \tau) = 0 \land (val(y(s), \tau) = 0 \lor val(y(s), \tau) = 1) \\ Poss(D(\tau), s) &\equiv val(y(s), \tau) = 1 \land (val(x(s), \tau) = 0 \lor val(x(s), \tau) = 1). \end{aligned}$$

Recall that τ is the time parameter added to the action at execution time by the interpreter. The function val evaluates continuous fluent expressions (cf. Chapter 4). For example, $val(linear(a_0, a_1, \tau_0), \tau)$ results in $a_0 + a_1 \cdot (\tau - \tau_0)$.

The preconditions of R and D mentioned fluent functions x and y. In the initial situation S_0, the agent is located in $(0, 1)$:

$$x(S_0) = constant(0) \quad y(S_0) = constant(1).$$

The X- and Y-coordinates are changed by the actions R and D, respectively. Their behavior is defined in terms of successor state axioms:

$$\begin{aligned} x(do(a, s)) = x_1 \equiv\ & \exists \tau, x_0 \,.\, a = R(\tau) \land x_0 = val(x(s), \tau) \land \\ & \quad x_1 = linear(x_0, 1\,\text{unit/s}, \tau) \lor \\ & \exists \tau \,.\, a \neq D(\tau) \land x_1 = constant(val(x(s), \tau)) \\ y(do(a, s)) = y_1 \equiv\ & \exists \tau, y_0 \,.\, a = D(\tau) \land y_0 = val(y(s), \tau) \land \\ & \quad y_1 = linear(y_0, -1\,\text{unit/s}, \tau) \lor \\ & \exists \tau \,.\, a \neq R(\tau) \land y_1 = constant(val(y(s), \tau)). \end{aligned}$$

Note that in the second part of each successor state axiom, the movement in the other direction must be stopped. For example, if the agent moves to the right in s and $a = D(\tau)$, then the agent just changed its direction from right to down and therefore the X-coordinate becomes constant.

A situation term $do([R(2), D(3)], S_0)$ means that at after 2 seconds, the agent started moving to the right which leads to $do(R(2), S_0)$. In this situation, the X-coordinate function is $x(do(R(2), S_0)) = linear(0, 1, 2)$. To determine whether or not action D is allowed at point in time 3, its precondition is tested:

$$\begin{aligned} Poss(D(3), do(R(2), S_0)) \equiv\ & val(y(do(R(2), S_0)), 3) = 1 \land \\ & (val(x(do(R(2), S_0)), 3) = 0 \lor \\ & \ val(x(do(R(2), S_0)), 3) = 1) \end{aligned}$$

$$\begin{aligned}
&\equiv val(constant(1), 3) = 1 \land \\
&\qquad (val(linear(0, 1, 2), 3) = 0 \lor \\
&\qquad\ val(linear(0, 1, 2), 3) = 1) \\
&\equiv 1 = 1 \land (0 + 1 \cdot (3 - 2) = 0 \lor 0 + 1 \cdot (3 - 2) = 1) \\
&\equiv 1 = 1 \land (1 = 0 \lor 1 = 1).
\end{aligned}$$

Therefore, the agent moves now down along the dashed edge towards $(1, 0)$. Since the agent moves with 1 unit/s, it must arrive at time 4 at the goal position $(0, 1)$: As shown above, the X-coordinate is 1 at time 3, and according to its successor state axiom, it stays constant from then on. The Y-coordinate, however, is then determined by the function $y(do([D(3), R(2)], S_0)) = linear(1, -1, 3)$. When point in time 4 is plugged in, this evaluates to $1 + (-1) \cdot (4 - 3) = 0$.

6.6.2 How the Interpreter Works

After having explained in detail how the agent moves and how the basic action theory models this movement, the rest of this subsection shows how the interpreter behaves when it faces some observations.

Assume that the agent has been seen at position $(0, 1)$ at point in time 1, at $(0.5, 1)$ at time 2.5 and at $(1, 0.5)$ at time 3.5. Obviously, this means that the agent moved along the dashed line. This results in the observation program

$$\begin{aligned}
\theta = {} & atomic(observe(1, x = 0 \land y = 1); waitFor(now > 1)); \\
& atomic(observe(2.5, x = 0.5 \land y = 1); waitFor(now > 2.5)); \\
& atomic(observe(3.5, x = 1 \land y = 0.5); waitFor(now > 3.5)).
\end{aligned}$$

Recall that the *waitFor* actions and the use of *atomic* prevent other actions from happening directly after an observation. Assume the interpreter wants to test whether or not the program

$$\sigma = R \parallel D$$

explains these observations. σ simply expresses that the agent moves either first right and then down, or first down and then right.

Recall that the interpreter resolves nondeterminism by choosing the alternative whose estimated reward is highest. Since there are no stochastic actions in $\sigma \parallel \theta$, the reward has not to be estimated in the present case. Instead, at a nondeterministic choice point, that one situation's branch is taken which has the highest reward.

Figure 6.5 depicts how *plan recognition by program execution* works. Reward function (6.5) counts the number of *observe* actions. Consider the top branch, $S_0 \to D(\tau_1) \to R(\tau_2) \to O_1 \to O_2$ where O_i stand for $atomic(observe(\ldots); waitFor(\ldots))$. This branch involves two observations, but the second observation fails. Hence, the situation in this branch with the best reward is reached when O_1 is executed. The reward is obviously 1.

The following takes a close look at how the interpreter executes the program $\sigma \parallel \theta$.

1. Starting in S_0, the interpreter needs to decide whether to move down, right or perform an observation. Moving down fails early, because the second observation sees the agent on the top edge at time 2.5. Moving right fails, too, because in order to entail the first observation, the agent would have to start moving at time 1, but then the second observation is violated. In both branches, the maximum encountered reward is 1. Since the third branch contains situations with higher reward (as can be seen in Figure 6.5), the interpreter executes $atomic(observe(1, x = 0 \wedge y = 1); waitFor(now > 1))$.

2. Again, the agent may move down, right or check for the next observation. Since the second observation cannot be entailed without moving, it is quickly ruled out. The interpreter may execute D immediately followed by R, thereby neutralizing the effect of the D action. Then, however, it cannot move down anymore and the third observation, which sees the agent on the right edge, fails. The maximum achievable reward is therefore 2 for this branch. The only remaining choice is the R action, which yields a superior reward of 3 as will be shown.

3. At this point, the interpreter has reached situation $do([O_1, R(\tau_1)], S_0)$ where O_1 denotes the first observation as in Figure 6.5. The remaining program is $D \parallel (O_2; O_3)$. Moving down now leads to a dead end, because then this move must happen before O_2, that is, $\tau_2 \leq 2.5$. This implies that the interpreter moves down along the left edge and O_2 cannot be executed. Consequently, O_2 is a better choice. In order to succeed, it constrains the timestamp of R to be $\tau_1 = 2$.

4. Finally, the agent needs to choose between D and O_3. Since the agent is still on the top edge and O_3 requires it to be in the middle of the right edge, D is executed first.

5. As last step, only O_3 remains. When $\tau_2 = 3$ is chosen as timestamp for the previous action D, the observation is entailed. The resulting situation's reward is 3 and better than any other situation's. Since the remaining program is empty, $\sigma \parallel \theta$ is executed completely.

The interpreter's agent has moved right on the top edge at time 2 and then issued an action to move downwards at time 3 along the right edge. This movement complies with the observations of the real agent.

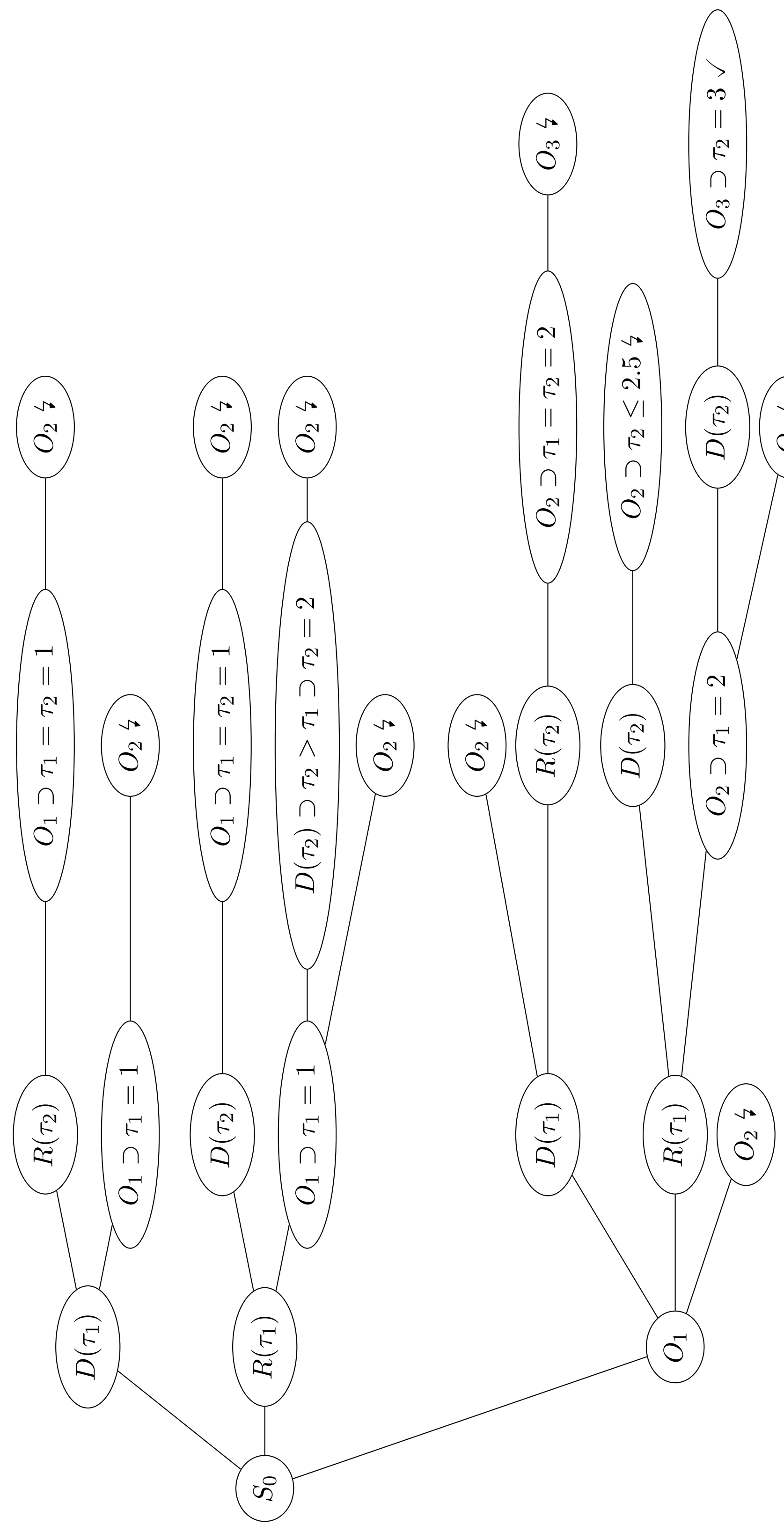


Figure 6.5: The decision tree of the interpreter while executing $\sigma \parallel \theta$. At each level, the nodes represent the different available actions. O_1, O_2, O_3 stand for the observations ($observe(1, x = 0 \wedge y = 1); waitFor(now > 1)$), ($observe(2.5, x = 0.5 \wedge y = 1); waitFor(now > 2.5)$), ($observe(3.5, x = 1 \wedge y = 0.5); waitFor(now > 3.5)$), respectively. A ↯ symbol means that the action is not possible in the current situation. τ_1, τ_2 are timestamps of R and D, which are constrained by monotonicity of time and preconditions, particularly by *observe* actions.

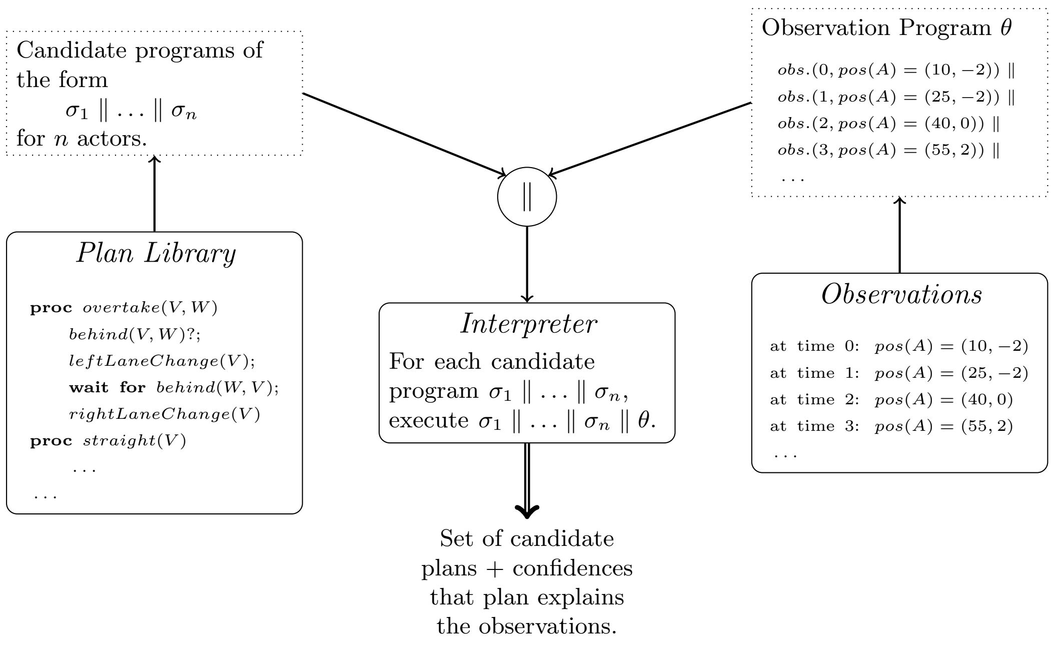

Candidate programs of the form $\sigma_1 \parallel \ldots \parallel \sigma_n$ for n actors.
Observation Program θ
$obs.(0, pos(A) = (10, -2)) \parallel$
$obs.(1, pos(A) = (25, -2)) \parallel$
$obs.(2, pos(A) = (40, 0)) \parallel$
$obs.(3, pos(A) = (55, 2)) \parallel$
...
$\parallel$
Plan Library
proc $overtake(V, W)$
$behind(V, W)?$;
$leftLaneChange(V)$;
wait for $behind(W, V)$;
$rightLaneChange(V)$
proc $straight(V)$
...
...
Interpreter
For each candidate program $\sigma_1 \parallel \ldots \parallel \sigma_n$, execute $\sigma_1 \parallel \ldots \parallel \sigma_n \parallel \theta$.
Observations
at time 0: $pos(A) = (10, -2)$
at time 1: $pos(A) = (25, -2)$
at time 2: $pos(A) = (40, 0)$
at time 3: $pos(A) = (55, 2)$
...
Set of candidate plans + confidences that plan explains the observations.

Figure 6.6: Refined structure of our approach to plan recognition. The initial structure depicted in Figure 1.1 on page 4 has been largely retained. Candidate programs are picked from the plan library and executed concurrently with the assembled observation program. Everything else happens inside the interpreter. Each candidate plan is not just assigned a yes/no-answer, but a numeric confidence instead.

6.7 Summary

The plan recognition system presented in this chapter originally belongs to the family of consistency-based approaches. A set of programs written in the language described in Chapter 5 constitutes the plan library. A plan is recognized by picking the respective program from the plan library and looking for an execution that matches all observations.

By the use of to stochastic actions, the system returns a confidence that a given program explains observations (cf. Section 6.5.1). Thus, the framework becomes a hybrid between the consistency-based and probabilistic approaches.

Decision theory makes the approach practical in that it provides a reasonable way to commit early to certain executions. While this heuristic destroys the algorithm's completeness, it keeps the otherwise evergrowing set of hypothesis at a constant size (cf. Section 6.3.3).

The system inherently models interdependencies between different actors; multi-agent plan recognition is easily done using concurrency (cf. Section 6.4).

The introduced *observe* actions gave rise to *plan recognition by program execution*; the whole plan recognition logic is expressed in Golog and therefore based on the situation calculus (cf. Section 6.3). This integrated approach proves to be very expressive. For one, it is the prerequisite for reasonable reward functions and thus also the basis for decision-theoretic resolution of nondeterminism and the aforementioned heuristic. For another, it allows to very naturally model sensor noise (cf. Section 6.5.2).

The general structure shown in Figure 1.1 on page 4 can be refined as done in Figure 6.6. The new figure resembles the introductory one very much: the most part of the plan recognition logic is bundled in the interpreter. The rest of the system simply needs to build the candidate programs and the observation program which are then executed concurrently by the Golog interpreter. In contrast to the initial sketch, the built system does not assign each candidate plan a yes/no-answer, but computes a numeric confidence that the plan explains the observations instead.

Chapter 7

Evaluation

Chapters 5 and 6 developed a framework for plan recognition. The following chapter documents the efforts to implement this system and to evaluate its performance.

The upcoming section returns to the modeling problem, this time having the defined plan recognition language in mind and looking for an efficient implementation. The subsequent section describes the interpreter, available software for constraint solving and a driving simulation. Finally, the experimental results of this testing environment are presented.

A comparison with other plan recognition systems is not part of this evaluation. For one, this is because the related approaches described in Section 2.1 seem not to be published as source code. For another, most alternative approaches have a different focus – many are about action hierarchies – and they are not suited for plan recognition in continuous domains.

7.1 Modeling Revisited

In reality, there is a close connection between the chosen class of constraints and modeling. This is because one needs to make a compromise between expressiveness and tractability of the constraint system. This aspect was ignored in Chapters 5 and 6 since the actual constraint solving is not a part of the theoretical system.

A related challenge is that the model should be abstract in order to allow for easy program formulation. On the other side, the model should precisely reflect the real physical behavior. Where the line between these demands should be drawn depends on the area of application.

A detailed analysis of a physical yet tractable driving model is beyond the scope of this thesis. One way to develop such a model would be to start off with a precise four-wheel model of a car and reduce this model until it is tractable. However, from a computational viewpoint it appears to be easier to start with a very simple model and enrich it when needed and possible. One argument for this way is that most published work related to constraint solving in continuous domains is about linear optimization problems (cf. Section 2.2). Hence, this proceeding is chosen in the following.

As in Chapter 4, a car is represented by a point in a two-dimensional Cartesian coordinate system. The Y-coordinate represents the lateral position, the X-coordinate is the longitudinal position.

In many scenarios, there is an infinite combination of action parameters and timestamps that could explain a single observation. For example, to achieve a speed of exactly 100 km/h at time 10 s in a world where acceleration is uniform and instantaneous, i.e. an action $setAccel(f, \tau)$ immediately changes the acceleration to f at τ, there is an infinite combination of (f, τ) tuples: the greater the acceleration f, the greater τ must be.

Values such as accelerations, yaws etc. are usually multiplied with time variables to determine the current speed or driven distance, for example. If the domain of the physical values such as acceleration is made finite and small, each constant value can be tried once. This drastically simplifies the resulting optimization problems.

The mechanisms for robustness presented in Chapter 6.5.1 work well with these finite domains, because they allow the reality to slightly differ from the chosen constant value.

Since velocities and yaws are restricted to finite domains, the simple modeling example from Section 4.1 only involves linear constraints. If instantaneous velocity changes are replaced with instantaneous acceleration changes as done in Section 4.3, the (in)equalities get quadratic.

7.2 Implementation

The proposed plan recognition system consists of a Golog interpreter with the semantics defined in Chapter 5 and a third-party constraint solver. For testing purposes, a racing game was used as a driving simulation.

7.2.1 Interpreter

The interpreter is written in ECLiPSe-CLP,[1] a constraint logic programming system which is largely compatible to Prolog.

A basic action theory can be defined in a module that exports the fluent predicates and some other predicates such as `primitive_action/1` and `stochastic_action/1` to declare primitive and stochastic actions, respectively.

[1]ECLiPSe-CLP is available at `http://www.eclipseclp.org/`.

The *transPr* function is implemented by means of sampling, because thus we only have to consider a single outcome situation per transition instead of all, potentially countably infinitely many new situations. The basic action theory needs to provide a predicate `random_outcome/3` that, given a stochastic action and a situation, returns a randomly chosen outcome action. The underlying probability distribution can be the same as in $prob_0$, but the probabilities are hidden from the caller.

The interpreter, which mainly consists of the predicates `trans_atom/4`, the analogue of *transAtPr*, and `trans/7`, the analogue of *transPr*, hence is ignorant towards probabilities. Instead, it picks a random outcome action at each stochastic action it encounters.

To determine the probability that a program can be executed, the interpreter is launched N times and the number of successful executions divided by N is then considered as the success probability. In particular, the probability that a program σ explains some observations represented by the observation program θ (cf. Section 6.3) is

$$\frac{\text{number of successful executions of } \sigma \parallel \theta}{\text{number of attempted executions of } \sigma \parallel \theta}.$$

A side benefit of sampling is easy parallelization. Since ECLiPSe-CLP has no support for multi-threading, this appears to be the only way to utilize the capacity of modern CPUs.

The implementation of the `trans/7` predicate is shown in Figure 7.1.

Intuitively, the execution probability should grow with increasing horizon. However, this is not always the case. The reason is a paradox in the implementation that at some point, the returned probability worsens with the horizon increase. The source of this counter-intuitive is probably the `random_outcome/3` predicate. Depending on the outcome action, executing a branch might fail during the lookahead. This may be either intended or it may be just due to an unlucky outcome action. Whatever the reason is, the branch is not be considered to resolve nondeterminism.

Consider the following basic action theory which illustrates the problem. Let there be two stochastic actions S and T with outcome actions A, B and F. S results with 90 % in A and 10 % in F. T results with 90 % in B and 10 % in F. While A and B are always possible, F's precondition always fails. Neither action has any effect. Assume the reward function returns the number of actions which are distinct from their predecessor if there is one. For example, the reward of $do(A, S_0)$ is 1, $do([A, B], S_0)$ is 2 and $do([A, A], S_0)$ is 1. For a program $S \parallel T$, the interpreter should behave the same way whatever the horizon is: if the last action was A, pick T in the hope of outcome B, otherwise pick S as next action hoping for A.

However, that is not the case, because the lookahead execution might fail. When `trans/7` is called with $S \parallel T$ in $do(A, S_0)$, the reward-maximizing choice of the next action is T. For horizon 1, the interpreter returns this choice in 90 % of all cases,

```
trans(RewardF, H, E, S, H1, E1, S1) :-
    next2(E, Ps),
    ( Ps = [] ->
        fail
    ; Ps = [(C, E1)] ->
        trans_atom(C, S, S1, C1),
        H1 is new_horizon(H, C1)
    ;
        ( param(RewardF, H, S),
          foreach((C, E), Ps),
          fromto((0, noop, nil), (H0, C0, E0), (H1, C1, E1), (H1, C, E1)),
          fromto((0, 0),         (LR0, LM0),   (LR1, LM1),   _) do
            ( test((    trans_atom(C, S, LS1, C2),
                        H2 is new_horizon(H, C2),
                        trans_max_h(RewardF, H2, E, LS1, _, LS2, LM2),
                        LR2 is reward(RewardF, LS2),
                        (   C0 == noop
                        ;   LR2 > LR0
                        ;   LR2 =:= LR0, LM2 > LM0)
                    ), (H2, C2, LR2, LM2))
            ->
                (H1, C1, E1) = (H2, C2, E),
                (LR1, LM1)   = (LR2, LM2)
            ;
                (H1, C1, E1) = (H0, C0, E0),
                (LR1, LM1)   = (LR0, LM0)
            )
        ),
        C \== noop,
        trans_atom(C, S, S1, _)
    ).
```

Figure 7.1: `trans/7` predicate implements *transPr* (5.11).
`RewardF` needs to be a functional fluent that returns a situation's reward. The lookahead horizon `H`, program `E` and situation `S` denote the current configuration, and `H1`, `E1` and `S1` are unified with the successor configuration.
`next2/2` unifies its second argument with a list of decompositions `(C, E1)` where `C` is a primitive, stochastic or test action and `E1` is the remaining program; this corresponds to $Next'$ (5.15). `C` is executed with `trans_atom/4` and `trans_max_h/7` performs the lookahead on `E1`. The reward-maximizing decomposition is chosen. The horizon is decremented by `new_horizon/3`.
`test/2` calls a goal with suppressed unification and extracts some variables which must have been bound to a ground term by the goal (here `H2`, `C2`, `LR2`, `LM2`). This avoids that constraints of potentially withdrawn lookahead branches are kept in the constraint pool.

Horizon	Probability	Runtime
1	0 %	4 s
2	10 %	36 s
3	63 %	223 s
4	78 %	972 s
5	73 %	3676 s

Table 7.1: Influence of different optimization horizons on the execution probability and on runtime of 100 independent interpreters. The test data are observations of a 15 s passing maneuver with 2 Hz.

because T leads to B with 90 %. If the horizon is 2, the chance that the interpreter chooses the reward-maximizing decomposition $B; A$ over $A; B$ can be derived from the following cases:

- if A fails during the lookahead of $A; B$, only B needs to succeed during the lookahead of $B; A$; the probability of this happening is $10\,\% \cdot 90\,\% = 9\,\%$,
- if A succeeds and then B fails during the lookahead of $A; B$, only B needs to succeed during the lookahead of $B; A$; the probability for this case is $90\,\% \cdot 10\,\% \cdot 90\,\% = 8.1\,\%$, and
- if both, A and B succeed during the lookahead of $A; B$, the whole branch $B; A$ needs to win over $A; B$; the probability for this event is $90\,\% \cdot 90\,\% \cdot 90\,\% \cdot 90\,\% = 65.61\,\%$.

The sum of these disjunct events and hence for `trans/7` resulting in $do([A, B], S_0)$ is 82.71 %. Experiments precisely confirm these numbers.

In fact, the implementation depicted in Figure 7.1 lessens the impact of this paradox. The reason is the more permitting behavior of `trans_max_h/7`. This predicate tries to perform `H2` many transitions, but it does not fail if the program fails before `H2` transitions. Instead, it unifies `LM2` with the number of transitions it performed. Then, `LR0`, `LR1`, `LR2` break the tie between branches with equal reward. However, this only reduces the effect, it does not solve the actual problem.

Table 7.1 shows how both, probability and runtime increase with the horizon. This roughly confirms the exponential development derived in Section 6.3.3.

Another trick in `trans/7` as depicted in Figure 7.1 is `new_horizon/3`. Intuitively, the horizon should be decremented by 1 after each transition. `new_horizon/3`, however, allows for a action-specific deduction of the horizon. For example, one might not want to decrease the horizon after *observe*, *waitFor* or test actions, since these have no effect on the world (except for shifting time). This allows for a middle course between two different global horizons.

One final question how to implement sampling regarding stochastic actions remains. During the lookahead, `trans/7` as depicted in Figure 7.1 unifies `C2` with the outcome action that was randomly chosen. This outcome action is memorized and finally executed. An alternative way would be to forget the outcome during the lookahead and pick a new outcome action which then triggers the actual transition. Both ways distort the probabilities.

For the first case, assume the program $(S \parallel True?); B$ should be executed. S again is a stochastic action with outcome A in 90 % and outcome F in 10 %, where A always succeeds and F always fails. $True?$ is a test action that always succeeds. The primitive action B is always possible, too, and the reward of $do(B, s)$ is 1 for any s and 0 otherwise. There are two competing execution orders: $S; True?; B$ and $True?; S; B$. Since both promise the same estimated reward 0.9, *transPr* picks any of them and executes it. The implementation in Figure 7.1, however, would achieve a reward of 1 in 99 % of all cases. This is because the `trans/7` predicate checks the execution order $S; True?; B$ first, which leads to $A; True?; B$ in 90 %. If this is the case, it memorizes A and thus gets to the rewarding situation. On the other hand, if S results in outcome F, the implementation chooses $True?$ first and then tries to execute $True?; S; B$. Here, S again leads to the succeeding outcome A with 90 %, which gives a total estimated reward of 1 by $90\,\% + 10\,\% \cdot 90\,\% = 99\,\%$. Without memorization of the outcome action, the interpreter would behave the correct way.

For the present domain, the following anomaly is perhaps more important. This anomaly only occurs if the interpreter forgets the atomic action of the best branch and chooses a new outcome action instead. Assume there is a program $(S \parallel (B_1; B_2)); B$, where S's outcome actions are again A and F with 90 % and 10 %, respectively. While B, B_1 and B_2 succeed always, A is only possible in situation $do(B_1, S_0)$. Again, $do(B, s)$ has reward 1 for all s, in all other situations, the reward is 0. When the interpreter reaches situation $do(B_1, S_0)$, it considers executing S during the lookahead. In 90 % of all cases, S leads to A and thus succeeds, otherwise the program execution eventually fails. Obviously, executing S is the best choice at this point. Hence, the interpreter decides to really execute S. However, since it needs to re-pick S's outcome action, the probability that S really leads to A in this transition is only $90\,\% \cdot 90\,\% = 81\,\%$.

The first anomaly mainly occurs when nondeterminism allows to execute different stochastic actions or a single stochastic action in different situations. The latter problem seems to be of more relevance for use cases such as passing maneuvers. The actions B_1 and B_2 can be identified with observation actions, and the stochastic action S may change lanes. In such a scenario, the observations predetermine the situation at which S should be executed.

Note that the source of these anomalies is the fact that `random_outcome/3` implements a simulation. The formal definition does not suffer from these problems.

Processes	COIN-OR CLP		ILOG CPLEX	
	Probability	Runtime	Probability	Runtime
1	72.6 %	2438 s	74.2 %	49 835 s
8	74.6 %	467 s	71.8 %	12 111 s

Table 7.2: Runtime of 1000 samples using different solvers on an Intel Core i7 CPU with four physical cores at 3.2 GHz and hyperthreading. The observations stem from a passing maneuver taking 15 s with two observations per second.

7.2.2 Constraint Solvers

ECLiPSe-CLP has two interfaces for constraint solving in continuous domains (Brisset et al., 2011): IC and EPLEX.

The library IC, interval constraints, is an integrated "general interval propagation solver" that supports integer and real variables and polynomial constraints (Brisset et al., 2011).

IC uses interval arithmetic to handle floating point inaccuracy (Brisset et al., 2011). Intervals are refined using a branch and prune algorithm described by Hentenryck et al. (1997). This algorithm uses interval arithmetic and Taylor extensions to find solutions. The algorithm is correct in that for each solution $(v_1, \ldots, v_n)$ of a polynomial system, there is one among the returned tuples of intervals $(I_1, \ldots, I_n)$ such that $v_i \in I_n$ (Hentenryck et al., 1997).

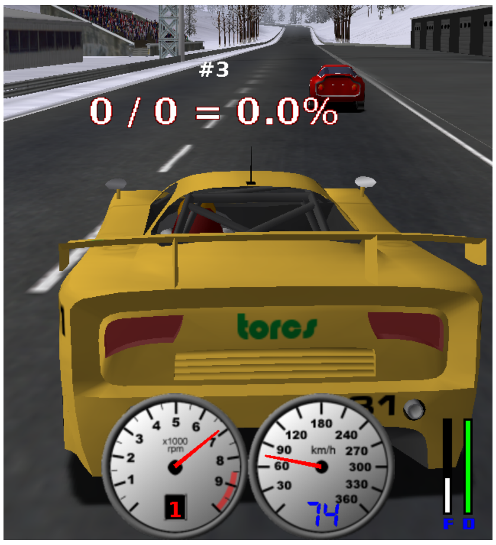
EPLEX is a generic interface to external constraint solvers for linear and mixed integer programs (Brisset et al., 2011). For this thesis, the commercial product IBM ILOG CPLEX Optimizer[2] and COIN-OR's open source solver CLP[3] were used.

The inherent intervals of IC proved to be a disadvantage. Similar to the initial interval-based approach to robustness sketched in Section 6.5.1, intervals become too large to make clear statements. The reason is simply that in interval arithmetic, the size of intervals grows exponentially with the number of multiplications, and the time variables are involved in many multiplications. Hence, the solution intervals need to be very small to keep them small, which greatly increases the runtime. On the plus side, IC is not limited to linear (in)equations.

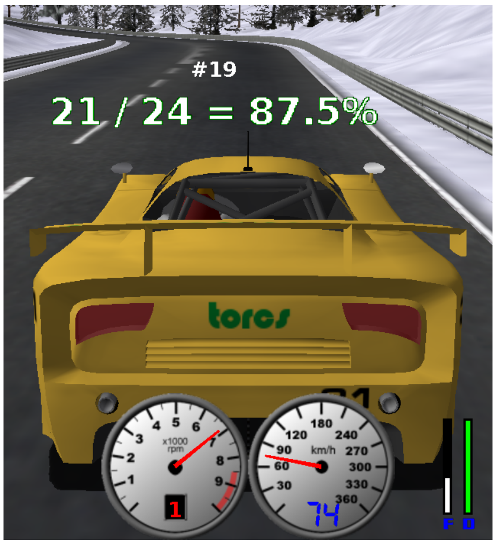
While the external solvers interfaced via EPLEX are limited to linear constraints, they provide exact solutions. Both, CLP and CPLEX are implementations of the Simplex algorithm. Surprisingly, COIN-OR's CLP outperformed ILOG CPLEX by far as (more than factor 20) shown in Table 7.2. The reason for the difference is not clear. It may be due to the fact that a preview version of CPLEX was used.

[2]IBM ILOG CPLEX Optimizer is available at `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[3]COIN-OR CLP and other solvers are available at `http://www.coin-or.org/`.



(a) Before a passing maneuver.



(b) After a passing maneuver.

Figure 7.2: Screenshots of TORCS with plan recognition output.

7.2.3 Driving Simulation

The computer racing game TORCS[4] is used as a driving simulation. Although there are no typical traffic elements such as intersections, racing tracks can be interpreted as highways. Snapshots from the game during a passing maneuver are shown in Figure 7.2.

Since the source code of TORCS is freely available, it is possible to modify the game and particularly implement custom AI drivers. For the specific case of a passing maneuver, an automatic driver was written. This driver's purpose is to be the overtaken vehicle. As such, the car stays straight on the right lane and keeps a specified speed.

The human driver can be controlled either by keyboard or a gaming steering wheel, for example. While the steering wheel gives a more realistic feeling and leads to smoother maneuvers, unfortunately it is more difficult to keep a car straight in the lane than a real car due to calibration problems.

The driving simulation generates observations in specific intervals. For the following tests, a frequency of 2 Hz was chosen, that is, TORCS sends an observation to the plan recognition system twice a second. Each observation consists of a pair of global longitudinal and a lateral coordinates of each car.[5] This is actually similar to GPS, except that there is no inaccuracy in TORCS.

[4]TORCS is available at `http://torcs.sourceforge.net/`.

[5]The lateral coordinate is basically the distance from the central line of the road. The longitudinal coordinate is measured from the finish line along the central line to the car's current position.

The plan recognition system follows the *plan recognition by program execution* paradigm from Section 6.3 by sampling. When the first observation occurs, the system spawns a fixed number of independent interpreters, each of which is initialized with the candidate programs (cf. the structural overview in Figure 6.6). From now on, each interpreter is fed the incoming observations.

All interpreters behave according to the heuristic plan recognition proposed in Section 6.3.3. This means that they always keep some observations buffered, because these observations are needed by the interpreter to reasonably resolve nondeterminism with a certain lookahead. Whenever enough observations are present, each interpreter triggers a transition until the program is final.

When an interpreter finishes execution, this is reported back to TORCS and displayed in the game. For example, the message "$0/0 = 0.0\,\%$" in Figure 7.2a means that up to now, none of the spawned interpreters has finished (successfully or unsuccessfully). In other words, all executions are consistent with the observations so far. The second picture, Figure 7.2b, is taken after the passing maneuver is completed. The displayed string "$21/24 = 87.5\,\%$" means that of 24 finished interpreters, 21 successfully executed the program representing a passing maneuver together with the observations. The resulting probability is $\frac{21}{24} = 87.5\,\%$.

Notice that the printed percentages stand for recognition of *completed* passing maneuvers. Online plan recognition is not enabled (even though one may draw conclusions from the number of interpreters not yet failed). The reason is simply that the probabilities of online plan recognition are not meaningful without the respective situation terms, which mirror how much of the maneuver has actually been performed yet.

If multiple candidate programs are tested, as in the second example in Section 7.3.2, the total number of interpreters is divided equally among the programs. For example, if two candidate programs are to be executed and the total number of interpreters is 24, then twelve interpreters are tasked with the first and the other twelve interpreters with the second program. This leads to two different probabilities, which are displayed separately on the screen.

For performance reasons, the interpreters run on a remote computer and their number is limited to 24 in the following examples. Surprisingly, this system allowed for real-time performance. Of course, with growing complexity of the model, performance quickly degrades, but on the other hand, the system can be easily scaled as long as a single interpreter can run on a single processor in real time.

7.3 Experimental Results

This section presents the experimental results of the prototypical implementation described in the previous section.

7.3.1 Passing Maneuver

In the following experiment, a car with (almost) constant speed of about 75 km/h overtakes another car driving (almost) constantly at 60 km/h on a two-lane road.

Test Setup

On one computer, the driving simulation TORCS (cf. Section 7.2.3) is installed. The slower car is controlled by TORCS to stay in the middle of the right lane at a constant speed of 60 km/h. A human being drives the overtaking car via keyboard or a steering wheel. The speed is limited by TORCS to 75 km/h to ensure that both cars move with almost constant speed.

This test case represents the example used throughout this thesis. The passing maneuver illustrates the requirements for the plan recognition system: It is inherently continuous and it requires a precise relationship between different vehicles, that is, it is not sufficient to consider just one actor. Furthermore, the points in time at which the overtaking car swings out and goes back into the lane are highly variable, which requires robustness towards time, and since the trajectory is not unique either, data robustness is crucial, too. Finally, many dangerous situations on freeways involve cars passing each other. Hence, the test case matches the requirements listed in Section 1.2 very well.

The candidate program is

$$\mathit{overtake}(V, W) \parallel \mathit{cruise}(W)$$

where *overtake* and *cruise* are defined in Figure 7.3. Note that in $\mathit{overtake}(V, W)$, only V acts actively, W just occurs in tests and thus remains passive. Even though the program $\mathit{cruise}(W)$ is utterly simple, this is multi-agent plan recognition as described in Section 6.4. As before, the presentation of the programs uses syntactic sugar such as **atomic** for the $\mathit{atomic}(\sigma)$ construct and **pick** for the operator $\pi v \,.\, \sigma$.

Recall that the third parameter of *setYaw* and *setVeloc* is used to parameterize the stochastic action's probability distribution. Here, log-normal probability distributions as depicted in Figure 6.2a are used. The third parameter specifies the random variable's standard deviation. Its unit is meters, because it eventually results in a tolerance area around the car. Hence, the underlying random variables are $X_1 \sim \log\mathcal{N}(-0.2, 0.7^2)$ while driving straight ahead and $X_2 \sim \log\mathcal{N}(-0.2, 1.0^2)$ during lane changes. This means that the lateral tolerance is about 1.0 m on average with a standard deviation of 0.8 m while driving straight. During the lane changes, the average is 1.3 m and the standard deviation 1.8 m. *setVeloc*'s probability distribution is $\log\mathcal{N}(1.0, 0.5^2)$.

```
**proc** $straightLeft(V)$
    **atomic**
        $setYaw(V, 0°, (-0.2\,\mathrm{m}, 0.7\,\mathrm{m}))$;
        $onLeftLane(V)$ **?**
    **endatomic**
**endproc**
```

```
**proc** $straightRight(V)$
    **atomic**
        $setYaw(V, 0°, (-0.2\,\mathrm{m}, 0.7\,\mathrm{m}))$;
        $onRightLane(V)$ **?**
    **endatomic**
**endproc**
```

```
**proc** $leftLaneChange(V)$
    **atomic**
        **pick** $\gamma \in \{4°, 6°, \ldots, 12°\}$ **do**
            $setYaw(V, \gamma, (-0.2\,\mathrm{m}, 1\,\mathrm{m}))$
        **endpick**;
        $onRightLane(V)$ **?**
    **endatomic**;
    $straightLeft(V)$
**endproc**
```

```
**proc** $rightLaneChange(V)$
    **atomic**
        **pick** $\gamma \in \{4°, 6°, \ldots, 12°\}$ **do**
            $setYaw(V, -\gamma, (-0.2\,\mathrm{m}, 1\,\mathrm{m}))$
        **endpick**;
        $onLeftLane(V)$ **?**
    **endatomic**;
    $straightRight(V)$
**endproc**
```

(a) Helper programs that keep the vehicle straight on a lane and change the lane, respectively. The **atomic** ensures that the primitive action and the test action are executed at once.

```
**proc** $overtake(V, W)$
    $behind(V, W)$ **?**;
    $onRightLane(V)$ **?**;
    $onRightLane(W)$ **?**;
    $straightRight(V)$;
    **begin**
        $leftLaneChange(V)$;
        **wait for** $behind(W, V)$;
        $rightLaneChange(V)$
    **concurrently with**
        $setVeloc(V, 75\,\mathrm{km/h}, (1\,\mathrm{m}, 0.5\,\mathrm{m}))$
    **end**;
    $onRightLane(W)$ **?**;
    $behind(W, V)$ **?**
**endproc**
```

```
**proc** $cruise(V)$
    $straightRight(V)$;
    $setVeloc(V, 55\,\mathrm{km/h}, (1\,\mathrm{m}, 0.5\,\mathrm{m}))$
**endproc**
```

(b) Top-level programs for a passing maneuver and the slower vehicle. The velocities are hard-coded in both programs.

Figure 7.3: Programs used in the passing maneuver example. All tuples are parameters for probability distributions underlying the stochastic actions.

The fact that both cars drive at nearly constant speed degrades the role of velocity in the whole system. Thus, all constraints are kept linear and therefore, a linear solver suffices. For this experiment, COIN-OR CLP is used.

In the implementation, the $setYaw$ action is slightly extended to merge the subsequent $onLeftLane(v)$ or $onRightLane(v)$ check, respectively. Hence, the programs $straightLeft$ and $straightRight$ boil down to a single primitive action. This is simply to improve performance and results because it increases the effective horizon without any additional cost.

Figure 7.4 visualizes some results of the plan recognition system applied to a completed passing maneuver. Each diagram corresponds to a successful execution of the aforementioned program by the interpreter. The figures only show the lateral position and tolerance, not the longitudinal position or tolerance of the car. This is sufficient for the present example as both vehicles' velocity is more or less constant.

Note that in Figure 7.4d, the lateral tolerance is that big that it allows the model to drive basically anywhere. Even though such high tolerances are rare due to the probability distribution, one may want to avoid them. In fact, they can be easily ruled out by restricting X_1, the random variable for lateral tolerances, not to be greater than 2.5 m, a half lane width. This is done in the implementation.

As said in Section 7.2.1, the program is executed multiple times. Each execution randomly picks one sample of each probability distribution involved in the program. For this reason, the tolerances differ between the different executions. Each result of the plan recognition consists of

- the probability that the program can be executed, that is, the quotient of successful and attempted executions, and
- the resulting situation term for each execution.

The probability may be interpreted as confidence that the chosen programs are executed by the different actors. However, this only half the deal; one still needs to consider the situation term. In Figure 7.5, (a)-(c) represent incomplete passing maneuvers. Yet, the interpreter successfully executes the program. Only (d) depicts a finished passing maneuver. By which degree the passing maneuver is executed, can be easily read off the situation term. For example, if the last action's timestamp is the same as the last *observe* action's timestamp, then the situation term corresponds to a complete passing maneuver.

The aforementioned probability distribution and its parameters are crucial to a good tradeoff between precision and recall. The log-normal distribution performed best in tests. The exponential distribution shown in Figure 6.2b proved to be difficult to parametrize in order to get reasonable results. The half-normal distribution, a special case of the folded normal distribution with $\mu = 0$, which in turn consists only of one half of the bell curve, leads to similarly well results but more false positives.

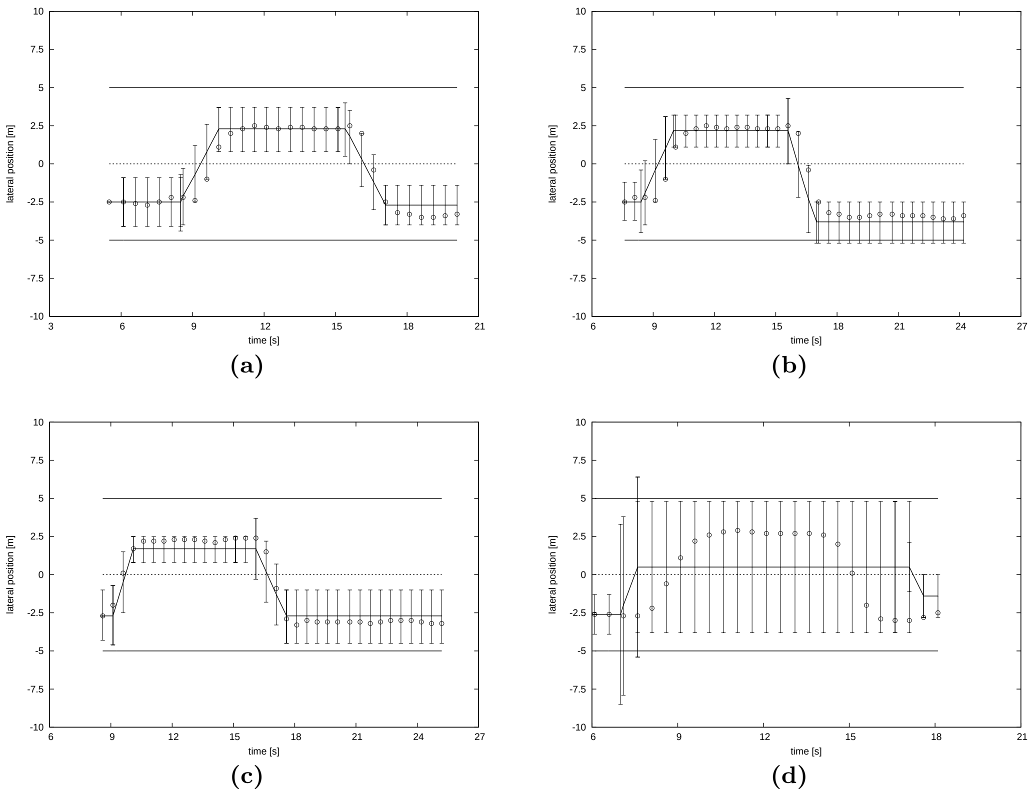


Figure 7.4: Visualization of the trajectory of an overtaking car and some samples of the Monte-Carlo simulation of different passing maneuvers. The X-axis represents the time, the Y-axis the lateral position. -5 to 0 is the right lane, 0 to 5 is the left lane. Points mark the observed lateral positions at the given time. The solid line follows the vehicle's trajectory in the model. Vertical lines represent the lateral tolerance which grow during lane changes.

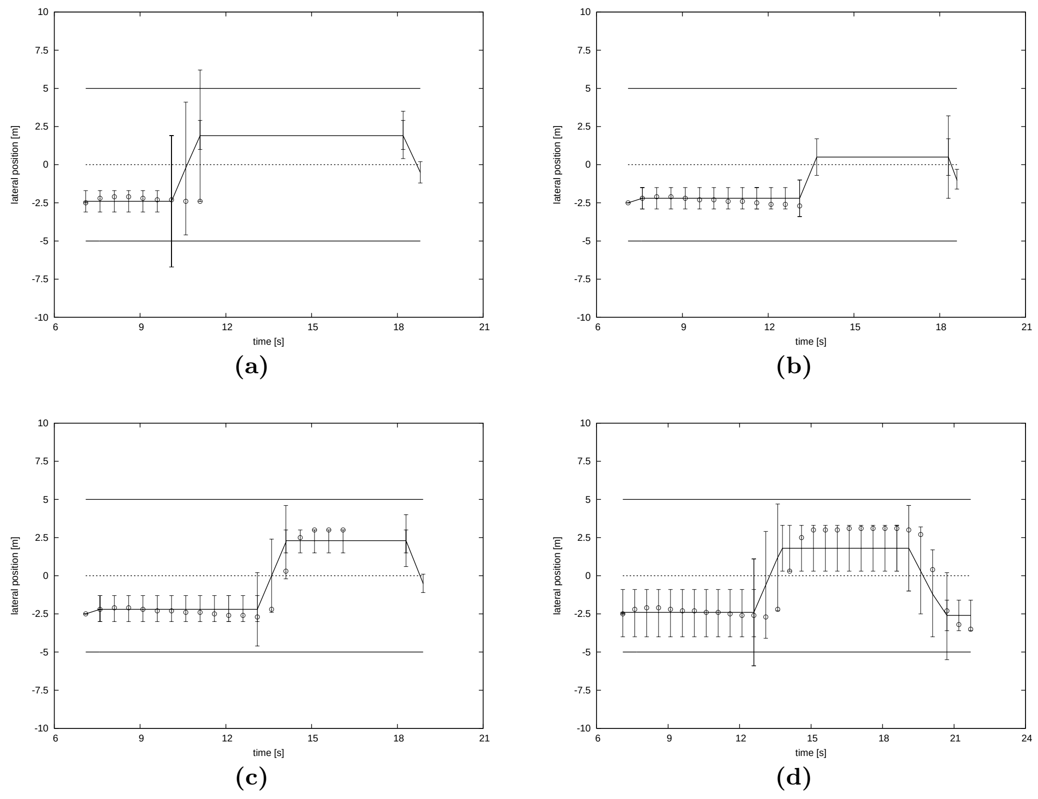


Figure 7.5: Visualization of the trajectory of an overtaking car during a single passing maneuver. Each subfigure represents the state of plan recognition at different points in time.

The log-normal distribution is an intuitive choice for the present problem for a number of reasons. For one, negative tolerances are ruled out. For another, a tolerance of 0 means that the driver exactly follows the model, which is physically impossible since instantaneous change of yaw etc. is a simplification of the real world in the model. The deviations from the model's idealized trajectory are therefore expected to be a skew bell curve distribution whose left side is much steeper than the right side. This is exactly the log-normal density function (cf. Figure 6.2a).

Note that the interpreter does not look for a execution that explains the observations *best*. This can be seen in Figure 7.5d: while the observations indicate that the vehicle has driven in the middle of the left lane, the model's trace is nearer to the center line of the road. The reason is that the interpreter randomly picks tolerances, and then only tries to find a way to drive so that all observations lie in the tolerance range. Fitting the model's trace better to the observations could be done with the least squares method, which in turn requires quadratic programming.[6]

Results

In an experiment, six human beings were asked to drive 20 maneuvers, 16 typical and legal passing maneuvers and four illegal[7] passing maneuvers on the right. The results are given in detail in Table 7.3. These numbers only refer to *completely* recognized passing maneuvers. If online plan recognition was enabled, the probabilities were meaningful only in connection with the respective situation term, which cannot be presented in a compact way.

This test's goal is not to achieve 100 % for each passing maneuver. The returned probability stands and falls with the parametrization of the probability distribution. Instead, the objective is to show that the plan recognition system is able to draw a line between passing maneuvers and non-passing maneuvers.

The first four test drivers used a steering wheel to control the virtual car. According to all of them, keeping the car straight in the lane is way more difficult than with a real car. This is mostly due to calibration issues of the steering wheel.

The fifth and sixth test driver used the keyboard. The keyboard control is choppier than the continuous control of the steering wheel. This matches the instantaneous movements of the model better, which is reflected in higher probabilities in Table 7.3.

The test succeeded insofar as the system clearly differentiated between legal and illegal passing maneuvers: each of the 96 legal passing maneuver resulted in a positive confidence, each of the 24 illegal ones yielded 0 %. With two exceptions, the returned probability for legal maneuvers was greater than 20 %. The fact that both exceptional

[6]COIN-OR CLP supports quadratic programming in general, but to build the objective function incrementally, a trick must be applied, which very badly affects the performance.

[7]Passing another vehicle on the right is not allowed in Germany.

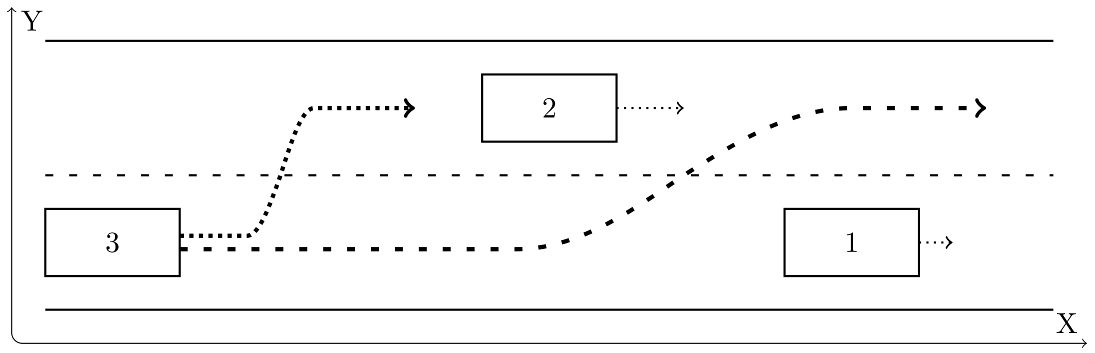


Figure 7.6: Alternative maneuvers to pass a car in the presence of a third car. While car 2 is attempting to overtake car 1, car 3 has two alternatives. To avoid a crash, it may either decelerate, get behind car 2 and slowly pass car 1 (dotted trace). Or it may continue to accelerate, pass car 2 (illegally) on the right side, change lanes between car 1 and 2 and thereby pass car 1 (dashed trace).

maneuvers were performed by the two most inexperienced drivers in the experiment suggests the conclusion that they are the result of unintended oscillating due to the calibration issues.

This test and other experiments showed that oscillating has the most significant impact on the returned probability. This is both reasonable and intended, because the more the driver oscillates in a lane, the higher lateral tolerances are needed.

By almost fixing both vehicles' speed, this experiment avoids a real world problem: nonlinear constraints. Due to the constant speed, only small longitudinal tolerances are needed. If more significant changes of velocity are expected, they also need to be modeled. An action to change the acceleration instantaneously appears to be more appealing than the instantaneous change of speed with $setVeloc$ (cf. Section 4.1 and Section 4.3).

7.3.2 Cautious versus Aggressive Passing

The second test case is about two alternative maneuvers. Consider a scenario of three cars driving in a two-lane road as depicted in Figure 7.6.

One car drives at 45 km/h in the right lane, the second car goes at 55 km/h in the fast lane. The third car rushes from behind in the right lane. It may either jam on the brakes, wait until the second car has passed the first one and then pass the first car on his part. Alternatively, the third car may try to aggressively weave its way between the first and second car without decelerating (which implies that it passes the second car illegally on the right).

This test's goal is to recognize which maneuver the third car performs.

	Person 1	Person 2	Person 3	Person 4	Person 5	Person 6
	29.2 %	20.8 %	8.3 %	16.7 %	45.8 %	54.2 %
	29.2 %	20.8 %	21.7 %	20.8 %	58.3 %	54.2 %
	29.2 %	29.2 %	29.2 %	25.0 %	58.3 %	54.2 %
	33.3 %	41.7 %	33.3 %	25.0 %	58.3 %	62.5 %
	33.3 %	45.8 %	33.3 %	29.2 %	66.7 %	66.7 %
	37.5 %	50.0 %	33.3 %	33.3 %	66.7 %	66.7 %
	41.7 %	50.0 %	37.3 %	33.3 %	70.8 %	70.8 %
	45.8 %	54.2 %	37.5 %	33.3 %	70.8 %	70.8 %
	45.8 %	54.2 %	37.5 %	37.5 %	75.0 %	75.0 %
	50.0 %	58.3 %	41.7 %	37.5 %	75.0 %	75.0 %
	62.5 %	62.5 %	45.8 %	41.7 %	75.0 %	75.0 %
	66.7 %	70.8 %	50.0 %	50.0 %	75.0 %	79.2 %
	66.7 %	75.0 %	54.2 %	50.0 %	79.2 %	79.2 %
	66.7 %	75.0 %	54.2 %	58.3 %	79.2 %	83.3 %
	70.8 %	83.3 %	54.2 %	70.8 %	83.3 %	91.7 %
	75.0 %	95.8 %	54.2 %	83.3 %	87.5 %	95.8 %
Avg.	49.0 %	55.5 %	38.9 %	40.4 %	70.3 %	72.1 %
St. Dev.	16.1 %	20.8 %	12.3 %	17.7 %	10.5 %	12.0 %
Avg.	54.4 %					
St. Dev.	20.2 %					

(a) Legal passing on the left.

	Person 1	Person 2	Person 3	Person 4	Person 5	Person 6
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
Avg.	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
St. Dev.	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

(b) Illegal passing on the right.

Table 7.3: Experiment of letting six different persons perform (a) legal and (b) illegal passing maneuvers.
The percentages are the probabilities returned by the plan recognition that the observations are be explained by a passing maneuver. They are given in ascending order per driver for readability. “Avg.” and “St. Dev.” stand for average and standard deviation, respectively.
The first four drivers used a steering wheel, while the fifth and sixth drivers used a keyboard.
The recurring values are due to the fact that each probability is a fraction with denominator 24 (cf. Section 7.2.3).

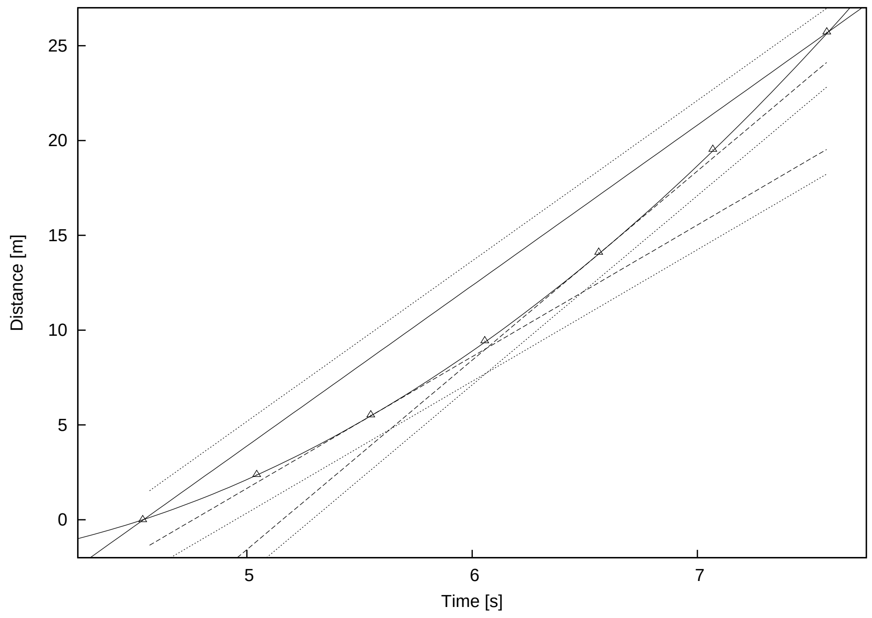


Figure 7.7: Linear approximation of the distance after driving 3 s with uniform acceleration of 3 m/s^2.
The crosses and the nonlinear solid line represent the actual acceleration; the linear solid line is the approximation for the interval from 4.5 s to 7.5 s.
The dotted lines are taken as upper and lower bounds. The upper bound is parallel to the approximation, whereas each lower bound is parallel to a tangent (dashed) of the actual acceleration function.

Linear Approximation of Uniform Acceleration

Since this example relies on velocity changes, a more sophisticated modeling is needed. The following introduces a way to represent acceleration with linear constraints, which is used for the present test case.

In Figure 7.7, the crosses stand for the measurements, which are matched very well by the solid nonlinear line that represents a uniform acceleration of 3 m/s^2. If this acceleration starts at 4.5 s and ends at 7.5 s, this can be approximated by the solid linear line.

Of course, the linear approximation does not match the observations as well as the nonlinear function. Hence, we need to state lower and upper bounds for the discrepancy between the model's and the measured acceleration. As with instantaneous actions, a tolerance constant is picked at random. An appropriate upper bound is hence the linear approximation shifted upwards by this constant.

The best lower bound would be the nonlinear function that describes the uniform acceleration (shifted downwards by the tolerance constant), which is obviously not possible in a linear system. However, the shape of this lower bound can be approximated by a sequence of linear tangents. Such a tangents are visualized by the dashed lines in Figure 7.7. Shifted by the negative of the tolerance constant, this yields one of the lower bounds (the bottom dotted line). Note that the space delimited by the upper bound and the lower bounds is a convex subspace and thus is suitable for a linear constraint solver.

This approximation scheme can be implemented with two actions, *startAccel* and *endAccel*. The former indicates that the vehicle begins to pick up a specified speed with a specified uniform acceleration, the latter marks the end of the acceleration process. Since the initial velocity is known, too, all physical values except for time are ground, which makes all involved functions linear and thus suitable for the linear constraint solver.

The simplified new preconditions are

$$\begin{aligned} Poss(startAccel(v, f, \tau), s) &\equiv \neg(\exists v', \tau') accelerating(v', \tau', s) \\ Poss(endAccel(\tau), s) &\equiv \exists v', \tau' \,.\, accelerating(v', \tau', s) \land \tau \leq \tau' \end{aligned}$$

where *accelerating* is a fluent that only holds during an acceleration process. Its first argument is the goal velocity, the second argument is the time at which the acceleration process would reach this velocity:

$$\begin{aligned} accelerating(v_1, \tau_1, do(a, s)) \equiv \exists v_0, f, \tau_0 \,.& \\ & a = startAccel(v_1, f, \tau_0) \land v_0 = val(veloc(s), \tau_0) \land \\ & \tau_1 = \tau_0 + (v_1 - v_0)/f \lor \\ & accelerating(v_1, \tau_1, s) \land (\forall \tau) a \neq endAccel(\tau). \end{aligned}$$

In reality, an additional parameter for the actor is added, and *startAccel* and *endAccel* are made stochastic actions to introduce a longitudinal tolerance.

For the sake of brevity, the new successor state axiom of *pos* and *veloc* are omitted. They are largely analogous to the original definition from Section 4.1. It should be noted, though, that *veloc* still behaves the same way as before except during acceleration processes, during which it returns $linear(v_0, f, \tau_0)$ such as the *veloc* fluent (cf. Section 4.3). Together with the precondition of *endAccel*, this allows to abort a running acceleration process and then still have a reasonable velocity in the model.

Test Setup

The technical test setup is very similar to the passing maneuver example. Again, the driving simulation TORCS (cf. Section 7.2.3) is used. Two computer-driven cars are ordered to drive at 45 km/h in the right lane and at 55 km/h in the fast lane. The second car starts about 5 s after the first one, so that a gap of initially 65 m between both cars continuously shrinks. It takes about 24 s for the gap to be closed.

With some delay, the human-controlled third car accelerates. In a first phase, the acceleration is about $3\,\mathrm{m/s^2}$. This phase takes about 4 s and expedites the car to about 54 km/h. The second phase again takes 4 s of $2\,\mathrm{m/s^2}$ acceleration to 83 km/h. Finally, the vehicle accelerates with $1\,\mathrm{m/s^2}$ for another 8 s to get to 115 km/h, which is then maintained.

The human driver may cancel this acceleration at any time to avoid a potential crash. In this case, the car is decelerated with about $-14\,\mathrm{m/s^2}$ to a speed of 52 km/h.

Note that all these physical values are only approximations. For example, the accelerations usually alternate by about $\pm 0.1\,\mathrm{m/s^2}$.

The acceleration phases are represented by a pair of *startAccel* and *endAccel* which are bundled in the helper program *accel* depicted in Figure 7.8a. The longitudinal tolerance is log-normally distributed $\log \mathcal{N}(1.0, 0.5^2)$ as in the previous example.

The lower and upper bounds are the linear approximations derived above. In our experiment, four tangents served as lower bounds.

The candidate programs for the two described maneuvers are given in Figure 7.8 for the third car. Note that the precondition of *endAccel* allows the user to cancel an acceleration process before it achieves its goal velocity. This is important in the *overtakeCautiously* program.

Hence, the alternative candidate programs are

$$\mathit{overtakeAggressively}(V_1, V_2, V_3) \parallel \mathit{cruise}_1(V_1) \parallel \mathit{cruise}_2(V_2)$$

and

$$\mathit{overtakeCautiously}(V_1, V_2, V_3) \parallel \mathit{cruise}_1(V_1) \parallel \mathit{cruise}_2(V_2)$$

where $\mathit{cruise}_{1,2}$ denote the variants of the *cruise* programs from Figure 7.3b with velocities 45 km/h and 55 km/h, respectively.

Half of the spawned interpreters are tasked with sampling the first candidate program, the other half deal with the second candidate program. As before, 24 interpreters are spawned in total, and observations are generated with 2 Hz.

As an example, Figure 7.9 shows the trace of a cautious maneuver. Vertical and horizontal bars represent the lateral and longitudinal tolerances, respectively. The growing distances between these bars reflect the fact that with growing velocity, the vehicle travels a greater distance between two observations. After about 325 m, the driver decelerates, which leads to lower velocity and thus smaller travelled distances.

```
**proc** $accel(V)$
    $startAccel(V, v, f, (1\,\mathrm{m}, 0.5\,\mathrm{m}))$;
    $endAccel(V, v, f)$
**endproc**
```

(a) Helper program that performs an acceleration process. Also consider the helper programs from Figure 7.3a.

```
**proc** $overtakeAggressively(V_1, V_2, V_3)$
    $straightRight(V_3)$;
    $accel(V_3, 54\,\mathrm{km/h}, 3\,\mathrm{m/s^2})$;
    $accel(V_3, 83\,\mathrm{km/h}, 2\,\mathrm{m/s^2})$;
    $accel(V_3, 115\,\mathrm{km/h}, 1\,\mathrm{m/s^2})$
**concurrently with**
    $waitFor(behind(V_3, V_1) \wedge$
            $behind(V_2, V_3))$;
    $leftLaneChange(V_3)$;
    $waitFor(behind(V_1, V_3) \wedge$
            $behind(V_2, V_3))$
**endproc**
```

```
**proc** $overtakeCautiously(V_1, V_2, V_3)$
    $straightRight(V_3)$;
    $accel(V_3, 54\,\mathrm{km/h}, 3\,\mathrm{m/s^2})$;
    $accel(V_3, 83\,\mathrm{km/h}, 2\,\mathrm{m/s^2})$;
    $accel(V_3, 115\,\mathrm{km/h}, 1\,\mathrm{m/s^2})$;
    $accel(V_3, 52\,\mathrm{km/h}, -14\,\mathrm{m/s^2})$;
    $leftLaneChange(V_3)$;
    $waitFor(behind(V_1, V_3) \wedge$
            $behind(V_3, V_2))$
**endproc**
```

(b) Top-level programs for a passing maneuver and the slower vehicle. The velocities are hard-coded in both programs.

Figure 7.8: Programs for lead-footed and cautious passing maneuvers, respectively. V_1 denotes the car in the right lane, V_2 the vehicle in the fast lane and V_3 represents the rushing car (cf. Figure 7.6).

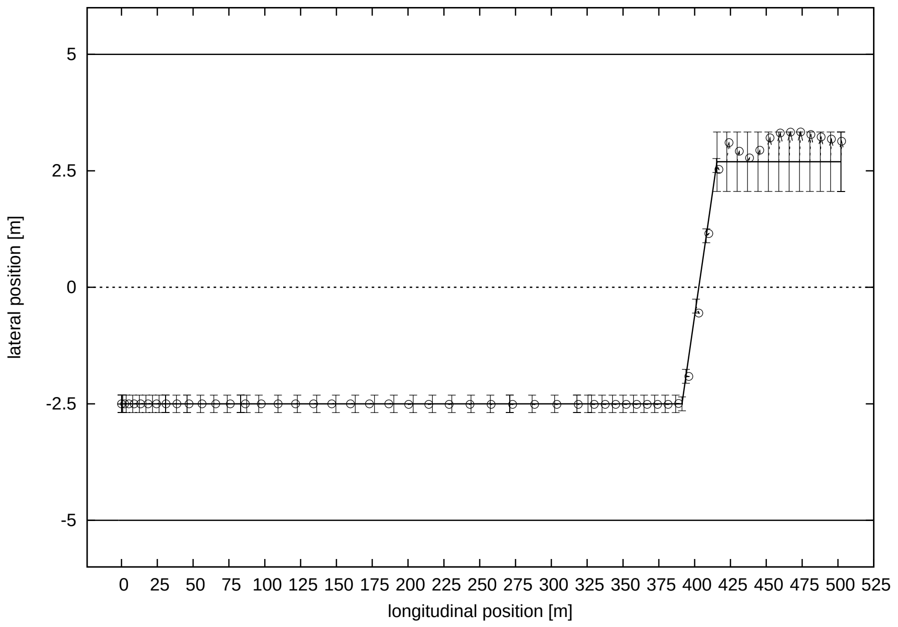


Figure 7.9: Visualization of acceleration and deceleration. The X- and Y-axis denote the longitudinal and lateral position, respectively. Points mark observed positions. Vertical bars represent lateral tolerances.

	Person 1				Person 2			
	Cautious		Aggressive		Cautious		Aggressive	
	Caut.	Aggr.	Caut.	Aggr.	Caut.	Aggr.	Caut.	Aggr.
	16.7 %	0.0 %	0.0 %	41.7 %	16.7 %	0.0 %	0.0 %	33.3 %
	16.7 %	0.0 %	0.0 %	50.0 %	25.0 %	0.0 %	0.0 %	50.0 %
	33.3 %	0.0 %	0.0 %	66.7 %	33.3 %	0.0 %	0.0 %	58.3 %
	33.3 %	0.0 %	0.0 %	66.7 %	33.3 %	0.0 %	0.0 %	58.3 %
	50.0 %	0.0 %	0.0 %	75.0 %	41.7 %	0.0 %	0.0 %	66.7 %
Avg.	30.0 %	0.0 %	0.0 %	60.0 %	30.0 %	0.0 %	0.0 %	53.3 %
St. Dev.	12.5 %	0.0 %	0.0 %	12.2 %	8.5 %	0.0 %	0.0 %	11.3 %

Table 7.4: Experiment of letting two persons perform five cautious and five aggressive maneuvers each.
The columns entitled “Caut.” contain the probabilities for the cautious candidate program, while “Aggr.” contains the results for the lead-footed program.
The recurring values are due to the fact that each probability is a fraction with denominator 12 (cf. Section 7.2.3).

Results

In an experiment, two test persons performed five cautious and five aggressive maneuvers. Both persons used the keyboard to control the virtual car. The plan recognition system is intended to recognize whether or not the driver behaved cautiously or aggressively.

Hence, when the test person brakes and slowly passes the objective car, the returned probabilities for the *overtakeCautiously* program should be greater than zero and those for *overtakeAggressively* should be zero. If, on the other hand, the driver aggressively passes the car at full throttle like the dashed line in Figure 7.6, the results should be the other way around.

The test results given in Table 7.4 confirmed this intended behavior in practice. Each maneuver was classified correctly in the sense that the correct maneuver was assigned a positive confidence while the incorrect one led to 0 %.

However, it is noticeable that the confidence in cautious maneuvers is about 30 % for both drivers, while the confidences in aggressive passing maneuvers are 20 % to 30 % higher. The reason is probably that the deceleration of $-14\,\mathrm{m/s^2}$ in the cautious program does not match the real deceleration as well as the accelerations do. An alternative or additional cause may be that a cautious passing maneuver takes longer than an aggressive one and the system therefore needs to explain more observations.

7.4 Summary

The proposed plan recognition system proved to be generally practical in the experiments. In both experiments, all maneuvers were classified as intended. Furthermore, the system surprised with real-time performance.

In the first example, the passing maneuvers with constant velocities, the lateral tolerances worked flawlessly. All maneuvers were recognized correctly in the sense that a program is considered as recognized iff the returned probability is positive. Longitudinal tolerances were almost needless due to the constant velocity.

The second example also required effective longitudinal tolerances, too. Though increasing the longitudinal tolerance solves this problem in a sense, this cuts down the significance of the system, because the longitudinal position is crucial to determine whether or not two cars crashed, for example. We tackled that problem with the *startAccel* and *endAccel* actions. However, this still requires that velocities and accelerations are hard-coded in the programs.

For practical reasons (e.g., availability and ECLiPSe-CLP interface), we decided upon a linear solver for our prototype. A more powerful constraint solver would eliminate the need for approximations in the model. For example, uniform acceleration could then be modeled accurately and without the limitation that the goal velocity must be known. In general, there is a tradeoff between the expressiveness of the constraint solver and the effort the axiomatizer has to put into simplifications and approximations. As it turned out, for the world of automotive traffic at least quadratic constraints would be very desirable.

However, considering the named limitations, our approach performed very satisfactory in all experiments.

Chapter 8

Conclusion and Future Work

This chapter concludes the thesis. It summarizes the findings and compares them to the requirements mentioned in Section 1.2. The second part identifies remaining problems to solve and possible future extensions.

8.1 Conclusion

This thesis proposes a new way for plan recognition in continuous domains such as automotive traffic. Given a candidate program from the predefined plan library, the system searches for an execution of this program which entails all observations. In contrast to many other plan recognition approaches which expect primitive actions to be observed directly, the proposed system supports arbitrary situation- and time-suppressed first-order formulas as observations.

A charming property of the system is that plan recognition boils down to program execution, because observations can be easily translated into a program that is executed concurrently with the candidate program. Thus, the semantics of the plan recognition procedure is entirely defined by the operational semantics of the program interpreter.

The system has its roots in the family of consistency-based approaches. This is the case in the literal sense as the logical conjunction of the program execution and the observations is required to be consistent. The efforts to introduce robustness using stochastic actions shift the focus towards the probabilistic direction. Thus, the proposed framework may be considered a hybrid between consistency-based and probabilistic plan recognition approaches.

At this point, the introductory demands should be compared to what the system delivers. In Section 1.2, continuous time, concurrency and robustness are listed as requirements. The following paragraphs show that the proposed plan recognition framework meets these requirements and provides some additional capabilities.

Continuous time is an inherent part of the semantics. The interpreter turns the qualitative descriptions of time in the programs into quantitative values at runtime. This bridges the opposing generality of programs and the concreteness of observations. It also introduces a kind of robustness, because programs do not specify timestamps uniquely. Instead, they constrain the choices of timestamps and the interpreter may choose according to these constraints *and* the observations.

Robustness is modeled by stochastic actions. Robustness is needed because things are less than perfect in the real world. In fact, there are three sources of uncertainty, namely

- the actors are not as perfect as the model, for example, when a car drives straight ahead, any human driver will in fact oscillate a bit and steadily control the car in order to keep the same direction,
- the model is not as perfect as the world, that is, not all physical influences cannot be represented in the model, and
- sensors are not definite but have measurement errors.

The first two kinds of robustness are represented in the programs from the plan library. In the passing maneuver program (cf. Figure 7.3), this was done with a tolerance area around the car. Sensor noise is modeled using stochastic actions, too: following the *plan recognition by program execution* approach, observations are actions themselves, which can be made stochastic and have outcome actions depending on the sensors' error models. Due to stochastic actions, the interpreter not just says whether or not the candidate program is consistent with the observations, but returns a probability that the program explains the observations. This probability can be considered the *confidence* that the certain program is executed.

One of the main contributions of this thesis is the integration of decision-theoretic Golog with concurrency, which led to the new semantics shown in Section 5.4.3. *Concurrency* is a key component of the plan recognition system at multiple points:

- programs may consist of concurrent subprograms themselves,
- *multi-agent* plan recognition is naturally done using concurrency, and
- concurrency is the prerequisite for the *plan recognition by program execution* paradigm.

Concurrency is thus the foundation of a number of features and should be worth the trouble of the new semantics.

Online plan recognition can be done in a natural fashion. Given a number of observations that only cover, say, half of the program, the interpreter executes this half in accordance with the observations. It returns the reached situation, which represents the executed half of the program, and the remaining half as remaining program. By this procedure, plans can be recognized at their beginning.

DTGolog-style decision theory is responsible for resolving nondeterminism in programs (which particularly includes concurrency). In combination with a lookahead horizon, this drastically improves the system's performance of program execution. This especially holds for online plan recognition, because the reward function can guide the interpreter a reasonable way through the program, even though the maneuver may not yet have been observed completely.

To sum up, Golog as modeling language and interpreter has been proven very powerful and surprisingly efficient. It supports a powerful notion of time, multi-agent and online plan recognition, and is robust towards data deviations and sensor noise. Despite this expressiveness, the prototype has performed very well, even in real time in some cases.

8.2 Future Work

The results of our experiments make this approach promising for future work.

As it turns out, the major limiting factor is the physical model. The more detailed the model, all the higher is the resulting mathematical constraints' complexity. While linear programs can be solved efficiently, they are probably overly simplistic, as they are not even capable of representing a uniform acceleration of a vehicle. A remaining challenge is therefore to find a more precise, yet tractable subclass of nonlinear programs. Section 2.2.2 mentioned some areas that are surely worth looking into. In the case of automotive traffic, changing from the global perspective to a vehicle-centered one might be help handling longitudinal tolerances.

Obviously, the required – and wanted! – degree of detail in the model depends on the area of application. In high level scenarios, the qualitative temporal and spatial representations from Section 2.2.1 might be an alternative to quantitative systems.

Valuing the probabilities returned by the plan recognition system is not easy. Though the general interpretation as confidence is clear, each probability strongly depends on the number of stochastic actions involved in the plan. Hence, some kind of normalization might be beneficial.

A remaining problem for real world plan recognition is when to start the interpreter. In the test scenarios, interpreters were spawned in the beginning. In reality, however, it is not known when a certain maneuver begins. The naive approach – continuously spawning interpreters – probably does not scale well.

Returning to the big picture outlined in the thesis' motivation, the ultimate goal is to predict critical situations. How plan recognition's result – a situation term, a remaining program and a confidence – can be extrapolated, still needs to be investigated.

Finally, plan recognition may be helpful in many other continuous areas such as domestic robotics and soccer robotics to early identify a human's intentions and the opponents' tactics, for example. It would be an interesting challenge to apply the presented method in these fields.

Bibliography

James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, November 1983.

Marcelo Gabriel Armentano. Approaches to plan recognition. Technical Report RR001-2005, Instituto de Sistemas Tandil, Universidad Nacional del Centro de la Provincia de Buenos Aires, 2005.

Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111(1-2):171–208, 1999.

Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, pages 355–362, Menlo Park, CA, July 2000a. AAAI Press.

Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Workshop on Decision-Theoretic Planning, Seventh International Conference on Principles of Knowledge Representation and Reasoning*, Breckenridge, Colorado, April 2000b.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, March 2004.

Pascal Brisset, Hani El Sakkout, Thom Frühwirth, Carmen Gervet Warwick Harvey, Micha Meier, Stefano Novello, Thierry Le Provost, Joachim Schimpf, Kish Shen, and Mark Wallace. *ECLiPSe 6.0 Constraint Library Manual*, 2011.

Hung H. Bui, Svetha Venkatesh, and Geoff West. Policy recognition in the abstract hidden markov model. *Journal of Artificial Intelligence Research*, 17:2002, 2002.

Sandra Carberry. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11:31–48, 2001.

Eugene Charniak and Robert Goldman. A probabilistic model of plan recognition. In *Proceedings of the ninth National conference on Artificial Intelligence*, volume 1 of *AAAI'91*, pages 160–165. AAAI Press, 1991.

Ismail Dagli, Michael Brost, and Gabi Breuel. Action recognition and prediction for driver assistance systems using dynamic belief networks. In *Proceedings of the Conference on Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, pages 179–194. Springer-Verlag, 2002.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

Alexander Ferrein, Stefan Schiffer, and Gerhard Lakemeyer. A fuzzy set semantics for qualitative fluents in the situation calculus. In *International Conference on Intelligent Robotics and Applications*, LNCS, pages 498–509, Wuhan, China, October 15-17 2008. Springer.

Christian Fritz and Sheila McIlraith. Decision-theoretic GOLOG with qualitative preferences. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 153–163, Lake District, UK, June 2006.

Christian Fritz and Sheila McIlraith. Computing robust plans in continuous domains. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 346–349, September 2009.

Christopher Geib and Robert Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173:1101–1132, 2009.

Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, June 2005.

Alexandra Goultiaeva and Yves Lespérance. Incremental plan recognition in an agent programming framework. In *Cognitive Robotics Workshop*, pages 83–90, 2006.

Henrik Grosskreutz. Probabilistic projection and belief update in the pGOLOG framework. In *European Conference on Artificial Intelligence*, 2000.

Henrik Grosskreutz and Gerhard Lakemeyer. cc-Golog: Towards more realistic logic-based robot controllers. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 2000a.

Henrik Grosskreutz and Gerhard Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *European Conference on Artificial Intelligence*, pages 548–552, 2000b.

Henrik Grosskreutz and Gerhard Lakemeyer. cc-Golog – an action language with continuous change. *Logic Journal of the IGPL*, 11(2):179–221, 2003.

Stephan Gspandl, Ingo Pill, Michael Reip, Gerald Steinbauer, and Alexander Ferrein. Belief management for high-level robot programs. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*. AAAI Press, to appear 2011.

Pascal Van Hentenryck, David McAllester, and Deepak Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34:797–827, 1997.

Henry A. Kautz and James F. Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, 1986.

Hector Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

Paulo Mateus, António Pacheco, Javier Pinto, Amílcar Sernadas, and Cristina Sernadas. Probabilistic situation calculus. *Annals of Mathematics and Artificial Intelligence*, 32:393–431, August 2001.

John McCarthy. Situations, Actions, and Causal Laws. Technical Report AI Memo 2 AIM-2, AI Lab, Stanford University, California, USA, July 1963. Published in Semantic Information Processing, ed. Marvin Minsky. Cambridge, MA: The MIT Press, 1968.

John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. First appearance of situation calculus allegedly in (McCarthy, 1963).

Daniel Meyer-Delius, Christian Plagemann, and Wolfram Burgard. Probabilistic situation recognition for vehicular traffic scenarios. In *Proceedings of the IEEE International Conference on Robotics and Automation*, ICRA'09, pages 4161–4166, Piscataway, NJ, USA, 2009. IEEE Press.

Hans-Hellmut Nagel and Michael Arens. Innervation des Automobils und Formale Logik. In Markus Maurer and Christoph Stiller, editors, *Fahrerassistenzsysteme mit maschineller Wahrnehmung*, pages 89–116. Springer Berlin Heidelberg, 2005.

Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

Miquel Ramirez and Hector Geffner. Plan recognition as planning. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

Miquel Ramirez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, 2010.

David Randell, Zhan Cui, and Anthony Cohn. A spatial logic based on regions and connection. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 165–176. Morgan Kaufmann, San Mateo, California, 1992.

Ray Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

Ray Reiter. Sequential, temporal GOLOG. In *Proceedings of Principles of Knowledge Representation and Reasoning*, pages 547–556, 1998.

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* The MIT Press, Massachusetts, MA, illustrated edition, 2001.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

Christoph Stiller, Sören Kammel, Irina Lulcheva, and Julius Ziegler. Probabilistische Methoden in der Umfeldwahrnehmung Kognitiver Automobile. *Automatisierungstechnik*, 56(11):563–574, 2008.

Tucker Taft and Robert Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *Lecture Notes in Computer Science*. Springer, 1997.

Nico Van de Weghe, Anthony G. Cohn, Philippe De Maeyer, and Frank Witlox. Representing moving objects in computer-based expert systems: the overtake event example. *Expert Systems with Applications*, 29:977–983, November 2005.

Nico Van de Weghe, Anthony Cohn, Guy De Tre, and Philippe De Maeyer. A qualitative trajectory calculus as a basis for representing moving objects in geographical information systems. *Control and Cybernetics*, 35(1):97–119, 2006.

Marc Vilain, Henry Kautz, and Peter Beek. Constraint propagation algorithms for temporal reasoning. In *Readings in Qualitative Reasoning about Physical Systems*, pages 377–382. Morgan Kaufmann, 1986.

Hans-Jürgen Zimmermann. *Operations Research.* Vieweg Verlag, 2005.