Plan Recognition by Program Execution in Continuous Temporal Domains

Christoph Schwering, Daniel Beck, Stefan Schiffer, and Gerhard Lakemeyer

Knowledge-based Systems Group, RWTH Aachen University, Aachen, Germany
(schwering,beck,schiffer,gerhard)@kbsg.rwth-aachen.de

Abstract. Much of the existing work on plan recognition assumes that actions of other agents can be observed directly. In continuous temporal domains such as traffic scenarios this assumption is typically not warranted. Instead, one is only able to observe facts about the world such as vehicle positions at different points in time, from which the agents' plans need to be inferred. In this paper we show how this problem can be addressed in the situation calculus and a new variant of the action programming language Golog, which includes features such as continuous time and change, stochastic actions, nondeterminism, and concurrency. In our approach we match observations against a set of candidate plans in the form of Golog programs. We turn the observations into actions which are then executed concurrently with the given programs. Using decision-theoretic optimization techniques those programs are preferred which bring about the observations at the appropriate times. Besides defining this new variant of Golog we also discuss an implementation and experimental results using driving maneuvers as an example.

1 Introduction

Much of the work on plan recognition, e.g. [1–5], has made the assumption that actions of other agents are directly observable. In continuous temporal domains such as traffic scenarios this assumption is typically not warranted. Instead, one is only able to *observe facts* about the world such as vehicle positions at different points in time, from which the agents' actions and plans need to be inferred. Approaches which take this view generally fall into the Bayesian network framework and include [6–8]. One drawback of these approaches is that actions and plans can only be represented at a rather coarse level, as the representations are essentially propositional and time needs to be discretized.

On the other hand, action formalisms based on first-order logic are very expressive and are able to capture plans at any level of granularity, including parallelism, continuous change and time. As we will see, this makes it possible to model the behavior of agents directly in terms of actions such as changing the direction of a vehicle or setting a certain speed. In a sense, this expressiveness allows to combine actions into plans or programs, whose execution can be thought of as an abstract *simulation* of what the agents are doing. This and parameterized actions yield a huge flexibility in formulating possible agent plans. Plan recognition in this framework boils down to finding those plans whose execution are closest in explaining the observed data.

In this paper, we propose an approach to plan recognition based on the action programming language Golog [9], which itself is based on the situation calculus [10, 11] and hence gives us the needed expressiveness. The idea is, roughly, to start with a plan library formulated as Golog programs and to try and match them online with incoming observations. The observations are translated into actions which can only be executed if the fact observed in the real world also is true in the model. These actions are executed concurrently with the given programs. Decision-theoretic optimization techniques are then used to select those among the modified programs whose execution bring about a maximum number of observations at just the right time.

Many of the pieces needed for a Golog dialect which supports this form of plan recognition already exist. These include concurrency [12], continuous change [13], stochastic actions [11], sequential time [14], and decision theory [15]. As we will see, these aspects need to be combined in novel ways and extended. The main contributions of the paper then are the definition of a new Golog dialect to support plan recognition from observations and to demonstrate the feasibility of the approach by applying it to traffic scenarios encountered in a driving simulator.[1] The rest of the paper is organized as follows. In the next section, we briefly outline our example traffic scenario. Section 3 introduces our new Golog variant prGolog, followed by a formal specification of an interpreter and a discussion of how plan recognition by program execution works in this framework. In Section 6, we present experimental results. Then we conclude.

2 Driving Maneuvers: An Example Domain

In this section we briefly introduce our example domain and some of the modeling issues it raises, which will motivate many of the features of our new Golog dialect.

In our car simulator a human controls a vehicle on a two-lane road with other cars controlled by the system. The goal is to recognize car maneuvers involving both the human-controlled car and others on the basis of observed global vehicle positions which are registered twice a second. For simplicity we assume complete knowledge and noise-free observations. We would like to model typical car maneuvers such as one vehicle passing another in a fairly direct and intuitive way. For that it seems desirable to build continuous time and continuous change directly into the modeling language. Among other things, this will allow us to define constructs such as $\mathsf{waitFor}(\mathsf{behind}(car1, car2))$, which lets time pass continuously until $car1$ is behind $car2$. To actually steer a car in the model, we will use actions to set the speed and to change the orientation (yaw). For simplicity and for complexity reasons, we assume that such changes are instantaneous and that movements are modeled by linear functions (of time) as in [13]. Concurrency comes into play for two reasons. For one, with multiple agents present they need

[1] We remark that the only other existing work using Golog for plan recognition [3] is quite different as it assumes that actions are directly observable.

Fig. 1: Two cars driving straight with different tolerances.

to be able to act independently. For another, observations will be turned into special actions which are executed concurrently with the agents' programs. Technically we will make use of ConGolog's notion of interleaved concurrency [12].

To see where probabilities come into play, we need to consider a complication which results from a mismatch between a simple model of driving in a straight line and reality, especially when a human controls a car. Most likely the human will oscillate somewhat even when his or her plan is to drive straight, and the amount of oscillation may vary over time and among individuals (see Figure 1 for two examples). Since the observed data will also track such oscillations, a straight-line model is not able to explain the data. Instead we introduce tolerances of varying width and likelihood, where the width indicates that a driver will deviate at most this much from a straight line and the likelihood estimates the percentage of drivers which exhibit this deviation. Technically, this means that the action which changes the direction of a car is considered a stochastic action in the sense of [15, 11]. We use a discretized log-normal distribution, where each outcome determines a particular tolerance. In a similar fashion, setting the speed introduces tolerances along the longitudinal axis to accommodate differences between the actual speed and the model.

3 The Action Language prGolog

prGolog is our new dialect of the action language Golog [9]. Golog is based on Reiter's version of the situation calculus [11] which is a sorted second-order language to reason about dynamic systems with actions and situations. A dynamic system is modeled in terms of a *basic action theory* (BAT) $\mathcal{D}$ which models the basic relationships of *primitive actions* and situation dependent predicates and functions, called *fluents*. A situation is either the initial situation S_0 or a term $do(a, s)$ where s is the preceding situation and a is an action executed in s. The main components of a BAT $\mathcal{D}$ are (1) precondition axioms $Poss(a, s) \equiv \rho$ that denote whether or not the primitive action a is executable in situation s, (2) successor state axioms which define how fluents evolve in new situations, and (3) a description of the initial situation S_0. A successor state axiom for a fluent $F(\boldsymbol{x}, s)$ has the form $F(\boldsymbol{x}, do(a, s)) \equiv \gamma_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s)$ where γ_F^+ and γ_F^- describe the positive and negative effects on fluent F, respectively.

Our simple model of a car consists of primitive actions that instantaneously change the vehicle's velocity and yaw, respectively. Furthermore, there are fluents $x(v, s)$ and $y(v, s)$ for the x and y-coordinates of the car v. Here, the x-axis points in the forward/backward direction and the y-axis in the left/right direction.

prGolog offers the same programming constructs known from other Golog dialects: deterministic and stochastic actions, test actions $\phi?$, sequences $\delta_1; \delta_2$, nondeterministic branch $\delta_1 \mid \delta_2$ and choice of argument $\pi v \,.\, \delta$, interleaved concurrency $\delta_1 \,\|\, \delta_2$, and others like if-then-else and while-loops, which are not needed in this paper. Also, to simplify the presentation, we use procedures as macros.

The prGolog programs in the plan library describe the plans an agent could be following. A lane change of a car v can be characterized as follows:

$$\begin{aligned}
&\textbf{proc } \mathsf{leftLaneChange}(v, \tau) \\
&\quad \pi\theta \,.(0^\circ < \theta \leq 90^\circ)?; \mathsf{waitFor}(\mathsf{onRightLane}(v), \tau); \mathsf{setYaw}(v, \theta, \tau); \\
&\qquad\qquad\qquad \pi\tau' \,.\, \mathsf{waitFor}(\mathsf{onLeftLane}(v), \tau'); \mathsf{setYaw}(v, 0^\circ, \tau').
\end{aligned}$$

This program leaves certain aspects of its execution unspecified. The angle θ at which the car v steers to the left may be nondeterministically chosen between 0° and 90°. While the starting time τ of the passing maneuver is a parameter of the procedure, the time τ' at which v gets back into the lane is chosen freely. The points in time are constrained only by means of the two waitFor actions in a way such that the car turns left when it is on the right lane and goes straight ahead when it is on the left lane. onRightLane and onLeftLane stand for formulas that specify what it means to be on the right and on the left lane, respectively. Using the procedure above an overtake maneuver can be specified as

$$\begin{aligned}
&\textbf{proc } \mathsf{overtake}(v, w) \\
&\quad \pi\tau_1 \,.\, \mathsf{waitFor}(\mathsf{behind}(v, w), \tau_1); \mathsf{leftLaneChange}(v, \tau_1); \\
&\quad \pi\tau_2 \,.\, \pi z \,.\, \mathsf{setVeloc}(v, z, \tau_2); \\
&\quad \pi\tau_3 \,.\, \mathsf{waitFor}(\mathsf{behind}(w, v), \tau_3); \mathsf{rightLaneChange}(v, \tau_3).
\end{aligned}$$

3.1 Stepwise Execution

To carry out plan recognition online, we will need to execute programs incrementally. ConGolog [12] introduced a transition semantics that does exactly this: a transition from a configuration (δ, s) to (δ', s') is possible if performing a single step of program δ in situation s leads to s' with remaining program δ'.

3.2 Time and Continuous Change

In the situation calculus, actions have no duration but are executed instantaneously. Hence, to get the position of a vehicle at a certain point in time, continuous fluents like $x(v, s)$ and $y(v, s)$ need to return *functions of time* which can be evaluated at a given time to get a concrete position. As in ccGolog [13], $y(v, s)$ returns a term $\mathit{linear}(a_0, a_1, \tau_0)$ which stands for the function of time $f(\tau) = a_0 + a_1 \cdot (\tau - \tau_0)$. The definition of successor state axioms for $x(v, s)$ and $y(v, s)$ to represent the effects of primitive actions is lengthy but straightforward.

We adopt sequential, temporal Golog's [14] convention that each primitive action has a timestamp parameter. Since these timestamped actions occur in situation terms, each situation has a starting time which is the timestamp of the last executed action. The precondition of a $\mathsf{waitFor}(\phi, \tau)$ action restricts the feasible timestamps τ to points in time at which the condition $\mathit{Poss}(\mathsf{waitFor}(\phi, \tau), s) \equiv \phi[s, \tau]$ holds. Here the syntax $\phi[s, \tau]$ restores the situation parameter s in the fluents in ϕ and evaluates continuous fluents at time τ. This precondition already captures the "effect" of waitFor, because just by occurring in the situation term, it shifts time to some point at which ϕ holds.

3.3 Stochastic Actions and Decision Theory

We include *stochastic actions* in prGolog which are implemented similarly to [11]. The meaning of performing a stochastic action is that nature chooses among a set of possible outcome actions. Stochastic actions, just like primitive actions, have a timestamp parameter. The setYaw action mentioned in the lane change program is a stochastic action. All outcomes for setYaw set the *yaw* fluent to the same value, they only differ in the width of the tolerance corridor described in Section 2 and Figure 1. In particular, the outcome actions are $\mathsf{setYaw}^*(v, \theta, \mathit{\Delta}, \tau)$ where $\mathit{\Delta}$ specifies the width of the tolerance corridor. Note that only the tolerance parameter $\mathit{\Delta}$ follows some probability distribution; the vehicle identifier v, the angle θ, and the timestamp τ are taken as they stand. We introduce a new fluent for the lateral tolerance, $\mathit{\Delta}y(v, s)$ whose value is the $\mathit{\Delta}$ of the last setYaw^* action. For $\mathsf{setVeloc}(v, z, \tau)$ we proceed analogously.

Stochastic actions introduce a second kind of uncertainty in programs: while nondeterministic features like the pick operator $\pi v\,.\,\delta$ represent choice points for the *agent*, the outcome of stochastic actions is chosen by *nature*. To make nondeterminism and stochastic actions coexist, we resolve the former in the spirit of DTGolog [15]: we always choose the branch that maximizes a *reward function*.

4 The Semantics of prGolog

For each program from the plan library we want to determine whether or not it explains the observations. To this end we resolve nondeterminism (e.g., concurrency by interleaving) decision-theoretically: when a nondeterministic choice point is reached, the interpreter opts for the alternative that leads to a situation s with the greatest reward $r(s)$. To keep computation feasible only the next l actions of each nondeterminstic alternative are evaluated. In Section 5 a reward function is shown that favors situations that explain more observations. Thus program execution reflects (observed) reality as closely as possible.

The central part of the interpreter is the function $\mathit{transPr}(\delta, s, l, \delta', s') = p$ which assigns probabilities p to one-step transitions from (δ, s) to (δ', s'). A transition is assigned a probability greater zero iff it is an optimal transition wrt reward function r and look-ahead l; all other transitions are assigned a probability of 0. *transPr* determines the optimal transition by inspecting all potential alternatives as follows: (1) compute all decompositions $\gamma; \delta'$ of δ where γ is a next *atomic action* of δ, (2) find a *best* decomposition $\gamma; \delta'$, and (3) execute γ. By *atomic action*, we mean primitive, test, and stochastic actions. A decomposition is considered *best* if no other decomposition leads to a higher-rewarded situation on average after l more transitions.

At first, we will define the predicate $\mathit{Next}(\delta, \gamma, \delta')$ that determines all decompositions $\gamma; \delta'$ of a program δ. We proceed with the function $\mathit{transAtPr}(\gamma, s, s') = p$ which holds if executing the atomic action γ in s leads to s' with probability p. Then, we define a function $\mathit{value}(\delta, s, l) = v$ which computes the estimated reward v that is achieved after l transitions of δ in s given that nondeterminism is resolved in an optimal way. *value* is used to rate alternative decompositions. With these helpers, we can define $\mathit{transPr}(\delta, s, l, \delta', s') = p$.

In our definition we often use **if** $\exists \boldsymbol{x} \,.\, \phi(\boldsymbol{x})$ **then** $\psi_1(\boldsymbol{x})$ **else** ψ_2 as a macro for $(\exists \boldsymbol{x} \,.\, \phi(\boldsymbol{x}) \wedge \psi_1(\boldsymbol{x})) \vee (\forall \boldsymbol{x} \,.\, \neg\phi(\boldsymbol{x}) \wedge \psi_2)$ where $\boldsymbol{x}$ is also visible in the then-branch.

4.1 Program Decomposition

$\mathit{Next}(\delta, \gamma, \delta')$ holds iff γ is a next atomic action of δ and δ' is the rest. It very much resembles ConGolog's *Trans* predicate except that it does not actually execute an action. Like ConGolog, we need to quantify over programs; for the details on this see [12]. Here are the definitions of *Next* needed for this paper:

$$\begin{aligned}
\mathit{Next}(\mathit{Nil}, \gamma, \delta') &\equiv \mathit{False} \\
\mathit{Next}(\alpha, \gamma, \delta') &\equiv \gamma = \alpha \wedge \delta' = \mathit{Nil} \quad (\alpha \text{ atomic}) \\
\mathit{Next}(\pi v \,.\, \delta, \gamma, \delta') &\equiv \exists x \,.\, \mathit{Next}(\delta^v_x, \gamma, \delta') \\
\mathit{Next}(\delta_1; \delta_2, \gamma, \delta') &\equiv \exists \delta_1' \,.\, \mathit{Next}(\delta_1, \gamma, \delta_1') \wedge \delta' = \delta_1'; \delta_2 \vee \\
&\qquad \mathit{Final}(\delta_1) \wedge \mathit{Next}(\delta_2, \gamma, \delta') \\
\mathit{Next}(\delta_1 \,\|\, \delta_2, \gamma, \delta') &\equiv \exists \delta_1' \,.\, \mathit{Next}(\delta_1, \gamma, \delta_1') \wedge \delta' = \delta_1' \,\|\, \delta_2 \vee \\
&\qquad \exists \delta_2' \,.\, \mathit{Next}(\delta_2, \gamma, \delta_2') \wedge \delta' = \delta_1 \,\|\, \delta_2'.
\end{aligned}$$

δ^v_x stands for the substitution of x for v in δ. $\mathit{Final}(\delta)$ holds iff program execution may terminate, e.g., for $\delta = \mathit{Nil}$. We omit it for brevity.

4.2 Executing Atomic Actions

Now we turn to executing atomic actions with *transAtPr*. *Test actions* are the easiest case because the test formula is evaluated in the current situation:

$$\mathit{transAtPr}(\phi?, s, s') = p \equiv \textbf{if } \phi[s] \wedge s' = s \textbf{ then } p = 1 \textbf{ else } p = 0.$$

Primitive actions have timestamps encoded as parameters like in sequential, temporal Golog, which are of the newly added sort real [14]. The BAT needs to provide axioms $\mathit{time}(A(\boldsymbol{x}, \tau)) = \tau$ to extract the timestamp τ of any primitive action $A(\boldsymbol{x}, \tau)$ and the function $\mathit{start}(\mathit{do}(a, s)) = \mathit{time}(a)$ which returns a situation's start time. The initial time $\mathit{start}(S_0)$ may be defined in the BAT. Using these, *transAtPr* can ensure monotonicity of time:

$$\begin{aligned}
&\mathit{transAtPr}(\alpha, s, s') = p \equiv \\
&\quad \textbf{if } \mathit{time}(\alpha[s]) \geq \mathit{start}(s) \wedge \mathit{Poss}(\alpha[s], s) \wedge s' = \mathit{do}(\alpha[s], s) \\
&\quad \textbf{then } p = 1 \textbf{ else } p = 0.
\end{aligned}$$

When a *stochastic action* β is executed, the idea is that nature randomly picks a primitive outcome action α. The axiomatizer is supposed to provide two macros $\mathit{Choice}(\beta, \alpha)$ and $\mathit{prob}_0(\beta, \alpha, s) = p$ as in [11]. The former denotes that α is a feasible outcome action of β, the latter returns the probability of nature actually choosing α in s. Probabilities are of sort real. The number of outcome actions must be finite. The axiomatizer must ensure that (1) any executable outcome action has a positive probability, (2) if any of the outcome actions is executable, then the probabilities of all executable outcome actions add up to 1, (3) no stochastic actions have any outcome action in common, and (4) primitive outcome actions do not occur in programs as primitive actions. The *transAtPr* rule returns the probability of the outcome action specified in s' if its precondition holds and 0 otherwise:

$$\begin{aligned}
&\mathit{transAtPr}(\beta, s, s') = p \equiv \\
&\quad \textbf{if } \exists \alpha, p' \,.\, \mathit{Choice}(\beta, \alpha) \wedge \mathit{transAtPr}(\alpha, s, s') \cdot \mathit{prob}_0(\beta, \alpha, s) = p' \wedge p' > 0 \\
&\quad \textbf{then } p = p' \textbf{ else } p = 0.
\end{aligned}$$

4.3 Rating Programs by Reward

The function *value* uses *transAtPr* to determine the maximum (wrt nondeterminism) estimated (wrt stochastic actions) reward achieved by a program. For a program δ and a situation s, *value* inspects the tree of situations induced by stochastic actions in δ up to a depth of look-ahead l or until the remaining program is final and computes the weighted average reward of the reached situations:

$$\begin{aligned}
&\mathit{value}(\delta, s, l) = v \equiv \\
&\quad \textbf{if } \exists v' \,.\, v' = \max_{\{(\gamma,\delta') \mid \mathit{Next}(\delta,\gamma,\delta')\}} \sum_{\{(s',p) \mid \mathit{transAtPr}(\gamma,s,s')=p \wedge p>0\}} p \cdot \mathit{value}(\delta', s', l-1) \wedge \\
&\qquad l > 0 \wedge (\mathit{Final}(\delta) \supset v' > r(s)) \\
&\quad \textbf{then } v = v' \textbf{ else } v = r(s).
\end{aligned}$$

The expression $\max_{\{(\gamma,\delta') \mid \mathit{Next}(\delta,\gamma,\delta')\}} f(\gamma, \delta') = v$ stands for

$$\exists \gamma, \delta' \,.\, \mathit{Next}(\delta, \gamma, \delta') \wedge v = f(\gamma, \delta') \wedge (\forall \gamma', \delta'')(\mathit{Next}(\delta, \gamma', \delta'') \supset v \geq f(\gamma', \delta'')).$$

For an axiomatization of the sum we refer to [16].

4.4 Transition Semantics

Finally, *transPr* simply looks for an optimal decomposition $\gamma; \delta'$ and executes γ:

$$\begin{aligned}
&\mathit{transPr}(\delta, s, l, \delta', s') = p \equiv \\
&\quad \textbf{if } \exists \gamma \,.\, \mathit{Next}(\delta, \gamma, \delta') \wedge \mathit{transAtPr}(\gamma, s, s') > 0 \wedge \\
&\qquad \big(\forall \gamma', \delta'' \,.\, \mathit{Next}(\delta, \gamma', \delta'') \supset \mathit{value}(\gamma; \delta', s, l) \geq \mathit{value}(\gamma'; \delta'', s, l)\big) \\
&\quad \textbf{then } \mathit{transAtPr}(\gamma, s, s') = p \textbf{ else } p = 0.
\end{aligned}$$

The function is consistent, i.e., $\mathit{transPr}(\delta, s, l, \delta', s')$ returns a unique p, for the following reason: If a primitive or a test action is executed, the argument is trivial. If a stochastic action β is executed, this is reflected in $s' = \mathit{do}(\alpha, s)$ for some primitive outcome action α and the only cause of α is β due to requirements (3) and (4). We will see that *transPr* is all we need for online plan recognition.

5 Plan Recognition by Program Execution

In our framework, plan recognition is the problem of executing a prGolog program in a way that matches the observations. An observation is a formula ϕ which holds in the world at time τ according to the sensors (e.g., ϕ might tell us the position of each car at time τ). For each of the, say, n vehicles, we choose a δ_i from the pre-defined programs as hypothetical explanation for the ith driver's behavior. These hypotheses are combined to a comprehensive hypothesis $\delta = (\delta_1 \parallel \ldots \parallel \delta_n)$ which captures that the vehicles act in parallel. We determine whether or not δ explains the observations. By computing a confidence for each explanation we can ultimately rank competing hypotheses.

To find a match between observations and program execution, we turn each observation into an action $\mathsf{match}(\phi, \tau)$ which is meant to synchronize the model with the observation. This is ensured by the precondition $\mathit{Poss}(\mathsf{match}(\phi, \tau), s) \equiv \phi[s, \tau]$ which asserts that the observed formula ϕ actually holds in the model at time τ. Hence, an executed match action represents an explained observation.

Plan recognition can be carried out online roughly by repeating two steps:
(1) If a new observation is present, merge the match action into the rest program.
(2) Execute the next step of the hypothesis program.
In practical plan recognition, it makes sense to be greedy for explaining as many observations as possible, with the ultimate goal of explaining all of them. This behavior can be easily implemented with our decision-theoretic semantics. Recall that the interpreter resolves nondeterministic choice points by opting for the alternative that yields the highest reward $r(s)$ after l further look-ahead steps. We achieve greedy behavior when we provide the reward function

$$r(s) = \text{number of } \mathsf{match} \text{ actions in } s.$$

While being greedy is not always optimal, this heuristic allows us to do plan recognition online. Since the interpreter can execute no more than l match actions during its look-ahead, nondeterminism is resolved optimally as long as the program contains at least l match actions. Thus, (2) is more precisely:
(2) If the program contains at least l match actions, execute the next step.

We now detail steps (1) and (2). Let δ be the hypothesis. The initial plan recognition state is $\{(\delta, S_0, 1)\}$ because, as nothing of δ has been executed yet, it may be a perfect hypothesis. As time goes by, δ is executed incrementally. However, the set grows because each outcome of a stochastic action must be represented by a tuple in the set.

Incoming observations are merged into the candidate programs by appending them with the concurrency operator. That is, when ϕ is observed at time τ we replace all configurations (δ, s, p) with new configurations $(\delta \,\|\, \mathsf{match}(\phi, \tau), s, p)$. When the number of match actions in δ is at least l, we are safe to update the configuration by triggering the next transition. Thus, upon matching the observation ϕ at time τ, a state $\mathcal{S}_i$ of the plan recognition evolves as follows:

$$\begin{aligned}\mathcal{S}_{i+1} = \{(\delta', s', p') \mid{}& (\delta, s, p) \in \mathcal{S}_i,\ \delta \text{ contains} \geq l-1 \text{ } \mathsf{match} \text{ actions},\\ &\mathcal{D} \cup \mathcal{C} \models p \cdot \mathit{transPr}(\delta \,\|\, \mathsf{match}(\phi, \tau), s, l, \delta', s') = p' \land p' > 0\}\\ \cup\, \{(\delta \,\|\, \mathsf{match}(\phi, \tau), s, p) \mid{}& (\delta, s, p) \in \mathcal{S}_i,\ \delta \text{ contains} < l-1 \text{ match actions}\}\end{aligned}$$

where $\mathcal{D}$ is a BAT and $\mathcal{C}$ are the axioms of our language. To simplify the presentation we assume complete information about the initial situation S_0.

Finally, we roughly describe how hypotheses can be ranked. Generally the idea is to sum the probabilities of those executions that explain the observations. By this means the hypothesis go_straight is ranked very well in Figure 1a, whereas the wide oscillations in Figure 1b cut off many of the likely but small tolerances. A complication arises because *transPr* does not commit to a single nondeterministic alternative if both are equally good wrt their reward. While our implementation simply commits to one of the branches which are on a par, *transPr* returns positive probabilities for all of them. With requirements (3) and (4) from Subsection 4.2 it is possible to keep apart these alternative executions. For space reasons we only sketch the idea: let $U_i \subseteq \mathcal{S}_i$ be a set of configurations (δ, s, p) that stem from *one* of the optimal ways to resolve nondeterminism. Then the confidence of U_i being an explanation so far is $\sum_{(\delta,s,p)\in U_i} p \cdot \frac{r(s)}{r(s)+m(\delta)}$ where $m(\delta)$ is the number of match actions that occur in the program δ. This weighs the probability of reaching the configuration (δ, s, p) by the ratio of explained observations $r(s)$ in the total number of observations $r(s) + m(\delta)$. Since there are generally multiple U_i, the confidence of the whole hypothesis is $\max_{U_i} \sum_{(\delta,s,p)\in U_i} p \cdot \frac{r(s)}{r(s)+m(\delta)}$.

6 Classifying Driving Maneuvers

We have implemented a prGolog interpreter and the online plan recognition procedure in ECLiPSe-CLP,[2] a Prolog dialect. We evaluated the system with a driving simulation, TORCS,[3] to recognize driving maneuvers. Our car model is implemented in terms of stochastic actions like setVeloc and setYaw and fluents like x and y which are functions of the velocity, yaw, and time. The preconditions of primitive actions, particularly of waitFor and match, impose constraints on these functions. For performance reasons we restrict the physical values like yaw and velocity to finite domains and allow only timestamps to range over the full floating point numbers so that we end up with *linear equations*. To solve these linear systems we use the constraint solver COIN-OR CLP.[4] The look-ahead to resolve nondeterministic choice points varies between two and three.

[2] `http://www.eclipseclp.org/`
[3] `http://torcs.sourceforge.net/`
[4] `http://www.coin-or.org/`

We modified the open source racing game TORCS for our purposes as a driving simulation. Twice a second, it sends an observation of each vehicle's noise-free global position (X_i, Y_i) to the plan recognition system. According to our notion of robustness, it suffices if the observations are within the model's tolerance. The longitudinal and lateral tolerance of each driver V_i is specified by the fluents $\Delta x(V_i)$ and $\Delta y(V_i)$ (cf. Section 3). Therefore, TORCS generates formulas of the form

$$\phi = \wedge_i \; |x(V_i) - X_i| \leq \Delta x(V_i) \wedge |y(V_i) - Y_i| \leq \Delta y(V_i).$$

Thus, the plan recognition system needs to search for possible executions of the candidate programs that match the observed car positions. If a smaller tolerance is good enough to match the observations, the confidence in the candidate program being an explanation for the observation is higher.

In our experiments, the online plan recognition kept the model and reality in sync with a delay of about two to five seconds. A part of this latency is inherent to our design: a delay of (look-ahead)/(observations per second) seconds is inevitable because some observations need to be buffered to resolve nondeterminism reasonably. This minimal latency amounts to 1.5 s in our setting, the rest is due to computational limitations.

6.1 Passing Maneuver

In our first scenario, a human-controlled car passes a computer-controlled car. To keep the equations linear, both cars have nearly constant speed (about 50 km/h and 70 km/h, respectively). Six test drivers drove 120 maneuvers in total, 96 of which were legal passing maneuvers (i.e., overtake on the left lane) and 24 were random non-legal passing maneuvers. We tested only one hypothesis which consisted of a program overtake for the human driver and a program go_straight for the robot car. Note that even though the robot car's candidate program is very simple, it is a crucial component because the passing maneuver makes no sense without a car to be passed. Hence, this is an albeit simple example of multi-agent plan recognition.

We encountered neither false positives nor false negatives: For all non-passing maneuvers the candidate program was rejected (confidence 0.0). In case the driver indeed did pass the robot car, our system valued the candidate program by a clearly positive confidence: 0.54 on average with standard deviation ± 0.2.

6.2 Aggressive vs Cautious Passing

In the second experiment, the human may choose between two ways to pass another vehicle in the presence of a third one as depicted in Figure 2. Robot car A starts in the right lane and B follows at a slightly higher speed in the left lane. The human, C, approaches from behind in the right lane with the aim to pass A. C may either continuously accelerate and attempt to aggressively pierce through the gap between B and A. Alternatively, if C considers the gap to be too small,

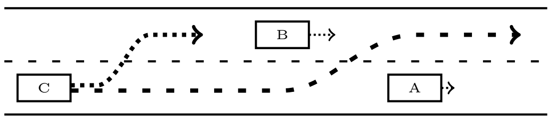

Fig. 2: While B passes A, C may choose between two maneuvers.

he or she may decelerate, swing out behind B, and cautiously pass A. To keep the equations linear, we approximate acceleration by incrementing the velocity in the model iteratively instead of setting it just once. Our system compares two competing hypotheses, one for C driving cautiously and one for the aggressive maneuver. The candidates for A and B are simply go_straight again. Note that although the programs for A and B are very simple, they are crucial because otherwise A and B would not move in the model.

We conducted this experiment 24 times with two different test drivers for C, each driving aggressively and cautiously in equal shares. When C behaved cautiously, this hypothesis was rated 0.3 on average (± 0.11) while the aggressive hypothesis was rated 0.0. When C drove aggressively, the aggressive program was rated 0.57 on average (± 0.12) and the cautious hypothesis was rejected with 0.0. Hence, the system distinguished correctly between the alternative hypotheses.

7 Discussion and Conclusion

In this paper, we proposed a new action language for specifying the behavior of multiple agents in terms of high-level programs. Among other things, the language combines decision theory to resolve nondeterminism with concurrency, and it supports temporal flexibility as well as robustness using stochastic actions.

On top of this language, we built online *plan recognition by program execution*. Observations are translated into match actions which are executed concurrently with candidate programs. Based on the decision-theoretic component and the transition semantics, a greedy heuristic, which preferred a maximal number of matched observations, worked well in our experiments.

The handling of continuous time and robustness distinguishes our approach from others like [1–5]. Neither of the approaches supports continuous time and change. [3, 4] also simulate candidate plans, but they require an action sensor, which is not given in continuous domains. Also, they do not provide any means to handle the mismatch between model and reality (cf. Section 2). While we use the plan library to reduce the space of explanations, [5] builds upon a pre-defined set of goals for which optimal plans (wrt a cost function) are computed and compared to the observed action sequence. This might lead to explanations that appear atypical to humans. We could achieve similar behavior with a program like $(\pi a \,.\, a)^*; \phi?$ which boils down to planning for goal ϕ. However, it is not clear whether or not [5] could handle fluent observations in continuous domains. Note that our approach also works if observations occur more sparsely than in our experiments – the program execution just needs to match fewer observations.

However, much more needs to be done to deal with real-world traffic scenarios. We believe that recognition can be improved with more realistic models of acceleration and the like. Also, qualitative models like QTC [17] should be considered. The assumption of complete information also needs to be relaxed. Finally, we are interested not only in recognizing plans but to predict potentially dangerous future situations to assist the driver.

References

1. Kautz, H.A., Allen, J.F.: Generalized plan recognition. In: Proc. of the Fifth Nat'l Conf. on Artificial Intelligence (AAAI'86). (1986) 32–37
2. Charniak, E., Goldman, R.: A probabilistic model of plan recognition. In: Proc. of the Ninth Nat'l Conf. on Artificial Intelligence (AAAI'91). (1991) 160–165
3. Goultiaeva, A., Lespérance, Y.: Incremental plan recognition in an agent programming framework. In Geib, C., Pynadath, D., eds.: Proc. of the AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR-07), AAAI Press (2007) 52–59
4. Geib, C., Goldman, R.: A probabilistic plan recognition algorithm based on plan tree grammars. Artificial Intelligence **173** (2009) 1101–1132
5. Ramirez, M., Geffner, H.: Plan recognition as planning. In: Proc. of the 21st Int'l Joint Conf. on Artificial Intelligence (IJCAI'09). (2009) 1778–1783
6. Pynadath, D.V., Wellman, M.P.: Accounting for context in plan recognition, with application to traffic monitoring. In: Proc. of the Eleventh Annual Conf. on Uncertainty in Artificial Intelligence (UAI'95), Morgan Kaufmann (1995) 472–481
7. Bui, H.H., Venkatesh, S., West, G.: Policy recognition in the abstract hidden markov model. Journal of Artificial Intelligence Research **17** (2002) 451–499
8. Liao, L., Patterson, D.J., Fox, D., Kautz, H.: Learning and inferring transportation routines. Artificial Intelligence **171** (2007) 311–331
9. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. J. Log. Program. **31** (1997) 59–84
10. McCarthy, J.: Situations, Actions, and Causal Laws. Technical Report AI Memo 2 AIM-2, AI Lab, Stanford University, California, USA (1963) Published in Semantic Information Processing, ed. Marvin Minsky. Cambridge, MA: The MIT Press, 1968.
11. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. The MIT Press (2001)
12. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121** (2000) 109–169
13. Grosskreutz, H., Lakemeyer, G.: cc-Golog – an action language with continuous change. Logic Journal of the IGPL **11** (2003) 179–221
14. Reiter, R.: Sequential, temporal GOLOG. In: Proc. of the Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 547–556
15. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proc. of the 17th Nat'l Conf. on Artificial Intelligence (AAAI'00), Menlo Park, CA (2000) 355–362
16. Bacchus, F., Halpern, J.Y., Levesque, H.J.: Reasoning about noisy sensors and effectors in the situation calculus. Artificial Intelligence **111** (1999) 171–208
17. Van de Weghe, N., Cohn, A.G., Maeyer, P.D., Witlox, F.: Representing moving objects in computer-based expert systems: the overtake event example. Expert Systems with Applications **29** (2005) 977–983