Cognitive Robotics
AAAI Technical Report WS-12-06

# Plan Recognition by Program Execution in Continuous Temporal Domains

**Christoph Schwering** and **Daniel Beck** and **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' intentions 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,[1] e.g. (Kautz and Allen 1986; Charniak and Goldman 1991; Goultiaeva and Lespérance 2007; Geib and Goldman 2009; Ramirez and Geffner 2009) 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' intentions need to be inferred. Approaches which take this view generally fall into the Bayesian network framework and include (Pynadath and Wellman 1995; Bui, Venkatesh, and West 2002; Liao et al. 2007). 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 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 is closest in explaining the observed data.

In this paper, we propose an approach to plan recognition based on the action programming language Golog (Levesque et al. 1997), which itself is based on the situation calculus (McCarthy 1963; Reiter 2001) 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 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 (De Giacomo, Lespérance, and Levesque 2000), continuous change (Grosskreutz and Lakemeyer 2003a), stochastic actions (Reiter 2001), sequential time (Reiter 1998), and decision theory in the spirit of DTGolog (Boutilier et al. 2000). As we will see, these aspects need to be combined in novel ways and extended. For example, DTGolog does not handle concurrency and it is intended to compute optimal policies in the sense of a Markov Decision Process. Here, however, we want to optimize wrt the given observations and explain as many of them as possible.[2] 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] In this paper we are only concerned with so-called keyhole plan recognition, where agents need not be aware that they are being observed.

[2] We remark that the only other existing work using Golog for plan recognition (Goultiaeva and Lespérance 2007) is quite different as it assumes that actions are directly observable.

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.

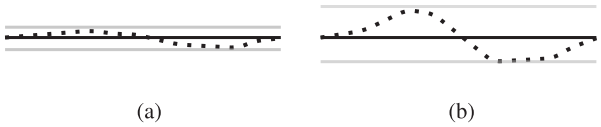
## 2 Driving Maneuvers: An Example Domain

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 $car2$. The definition will be a variant of the waitFor-construct defined in (Grosskreutz and Lakemeyer 2003a). 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 (Grosskreutz and Lakemeyer 2003a). Concurrency comes into play for two reasons. For one, with multiple agents present they need 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 (De Giacomo, Lespérance, and Levesque 2000).

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 intention 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 unless we allow for an arbitrary number of changes of direction, which is cumbersome and computationally expensive. 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 (Boutilier et al. 2000; Reiter 2001). We use a discretized log-normal distribution,



(a) (b)

Figure 1: Two cars driving straight with different tolerances.

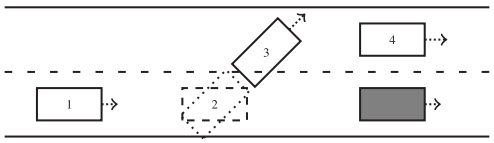



Figure 2: A passing maneuver.

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.

With this idea of tolerances in mind it will become rather unlikely that an actual passing maneuver such as the one depicted in Figure 2, where the white car is observed at positions 1, 3, and 4, is still considered driving straight. Instead, executing a program modeling a passing maneuver will be considered a better match, as it includes a change of orientation at, say, position 2.

## 3 The Action Language prGolog

prGolog is our new dialect of the action language Golog (Levesque et al. 1997). Golog is based on Reiter's version of the situation calculus (Reiter 2001) 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 $F(\vec{x}, s)$ has the form $F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$ where $\gamma_F^+$ and $\gamma_F^-$ 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 choice of argument $\pi v \,.\, \delta$ and nondeterministic branch $\delta_1 \,|\, \delta_2$, interleaved concurrency $\delta_1 \parallel \delta_2$, and others such as

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 different plans an agent could be following. For instance, a lane change of a car $v$ can be characterized as follows:

**proc** leftLaneChange$(v, \tau)$
$\pi\theta\,.(0^\circ < \theta \leq 90^\circ)?;$
waitFor(onRightLane$(v), \tau)$; setYaw$(v, \theta, \tau)$;
$\pi\tau'\,.$ waitFor(onLeftLane$(v), \tau')$; setYaw$(v, 0^\circ, \tau')$
**endproc**.

This program leaves certain aspects of its execution unspecified. The angle $\theta$ at which the car $v$ steers to the left may be nondeterministically chosen between $0^\circ$ and $90^\circ$. While the starting time $\tau$ of the passing maneuver is a parameter of the procedure, the time $\tau'$ 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

**proc** overtake$(v, w)$
$\pi\tau_1\,.$ waitFor(behind$(v, w), \tau_1)$; leftLaneChange$(v, \tau_1)$;
$\pi\tau_2\,.\,\pi z\,.$ setVeloc$(v, z, \tau_2)$;
$\pi\tau_3\,.$ waitFor(behind$(w, v), \tau_3)$; rightLaneChange$(v, \tau_3)$
**endproc**.

### 3.1 Stepwise Execution

To carry out plan recognition online, we will need to execute programs incrementally, one action at a time. ConGolog (De Giacomo, Lespérance, and Levesque 2000) introduced a transition semantics that does exactly this: a transition from a configuration $(\delta, s)$ to a configuration $(\delta', s')$ is possible if performing a single step of program $\delta$ in situation $s$ leads to a new situation $s'$ with remaining program $\delta'$.

### 3.2 Time and Continuous Change

In the situation calculus, actions have no duration but are executed instantaneously. Therefore we model continuous fluents (e.g., a car's position) by fluent functions each of which returns a *function of time*. E.g., the function returned by $x(v, s)$ denotes the longitudinal movement of $v$ in $s$. This function can be evaluated at a given point in time to obtain a concrete position. As in ccGolog (Grosskreutz and Lakemeyer 2003a), we model these functions of time with terms like $linear(a_0, a_1, \tau_0)$ which stands for the function $f(\tau) = a_0 + a_1 \cdot (\tau - \tau_0)$. 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 (Reiter 1998) 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 waitFor$(\phi, \tau)$ action restricts the feasible timestamps $\tau$ to points in time at which the condition $\phi$ holds:

$$Poss(\mathsf{waitFor}(\phi, \tau), s) \equiv \phi[s, \tau]$$

where the syntax $\phi[s, \tau]$ restores the situation parameter $s$ in the fluents in $\phi$ and evaluates continuous fluents at time $\tau$. This precondition already captures the "effect" of waitFor, because just by occurring in the situation term it shifts time to some point at which $\phi$ holds. Unlike ccGolog, we do not enforce a least time point semantics for added flexibility.

### 3.3 Stochastic Actions and Decision Theory

We include *stochastic actions* in prGolog which are implemented similarly to (Reiter 2001). 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 setYaw$^*(v, \theta, \Delta, \tau)$ where $\Delta$ specifies the width of the tolerance corridor. Note that only the additional parameter $\Delta$ follows some probability distribution; the vehicle identifier $v$, the angle $\theta$, and the timestamp $\tau$ are taken as they stand. We introduce a new fluent for the lateral tolerance, $\Delta y(v, s)$ whose value is the $\Delta$ of the last setYaw$^*$ action. Analogously, we add longitudinal tolerance: setVeloc$(v, z, \tau)$ is stochastic and has outcome actions setVeloc$^*(v, z, \Delta, \tau)$. The longitudinal tolerance is captured by the new fluent $\Delta x(v, s)$.

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 by always choosing the branch that maximizes a *reward function* as in DTGolog (Boutilier et al. 2000). In contrast to DTGolog, we do not compute a policy. To combine decision theory and concurrency, we propose a new transition semantics in the next section.

## 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 $transPr(\delta, s, l, \delta', s') = p$ which assigns probabilities $p$

to one-step transitions from $(\delta, s)$ to $(\delta', s')$. 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. That is, we specify a transition semantics similar to ConGolog (De Giacomo, Lespérance, and Levesque 2000) that integrates decision-theoretic ideas from DTGolog (Boutilier et al. 2000). $transPr$ determines the optimal transition by inspecting all potential alternatives as follows:

(1) compute all possible decompositions $\gamma; \delta'$ of $\delta$ where $\gamma$ is a next *atomic action* of $\delta$,

(2) find a *best* decomposition $\gamma; \delta'$, and

(3) execute $\gamma$.

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.

At first, we will define the predicate $Next(\delta, \gamma, \delta')$ that determines all decompositions $\gamma; \delta'$ of a program $\delta$. We proceed with the function $transAtPr(\gamma, s, s') = p$ which holds if executing the atomic action $\gamma$ in $s$ leads to $s'$ with probability $p$. Then, we define a function $value(\delta, s, l) = v$ which computes the estimated reward $v$ that is achieved after $l$ transitions of $\delta$ in $s$ given that nondeterminism is resolved in an optimal way. $value$ is used to rate alternative decompositions. With these helpers, we can define $transPr(\delta, s, l, \delta', s') = p$.[3]

We often use the following if-then-else macro where the quantified variables are also visible in the then-branch:

$$\textbf{if } \exists \vec{x} \,.\, \phi(\vec{x}) \textbf{ then } \psi_1(\vec{x}) \textbf{ else } \psi_2 \stackrel{\text{def}}{=} (\exists \vec{x} \,.\, \phi(\vec{x}) \wedge \psi_1(\vec{x})) \vee (\forall \vec{x} \,.\, \neg\phi(\vec{x}) \wedge \psi_2)$$

## 4.1 Program Decomposition

$Next(\delta, \gamma, \delta')$ holds iff $\gamma$ is a next atomic action of $\delta$ and $\delta'$ is the rest. It very much resembles ConGolog's $Trans$ predicate except that it does not actually execute an action.[4] Like ConGolog, we need to quantify over programs; for the details on this see (De Giacomo, Lespérance, and Levesque 2000). Here are the definitions needed for $Next$:

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

[3] pGOLOG's (Grosskreutz and Lakemeyer 2003b) $transPr$ is much more limited as it does not account for nondeterminism.

[4] We remark that we differ from the $next$ defined in (De Giacomo, Lespérance, and Muise 2011) in that our $Next$ is not functional.

$Final(\delta)$ holds iff program execution may terminate. For instance, the empty program $Nil$ is final, primitive actions are never final, and a sequence $\delta_1; \delta_2$ is final iff $\delta_1$ and $\delta_2$ are. In general, our $Final$ is the same as ConGolog's except that it has no situation argument:[5]

$$\begin{aligned}
Final(Nil) &\equiv True \\
Final(\alpha) &\equiv False \ \ (\alpha \text{ atomic}) \\
Final(\pi v \,.\, \delta) &\equiv \exists x \,.\, Final(\delta_x^v) \\
Final(\delta_1 \mid \delta_2) &\equiv Final(\delta_1) \vee Final(\delta_2) \\
Final(\delta_1; \delta_2) &\equiv Final(\delta_1) \wedge Final(\delta_2) \\
Final(\delta_1 \parallel \delta_2) &\equiv Final(\delta_1) \wedge Final(\delta_2) \\
Final(\delta^*) &\equiv True.
\end{aligned}$$

## 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:

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

*Primitive actions* have timestamps encoded as parameters like in sequential, temporal Golog, which are of the newly added sort real (Reiter 1998). The BAT needs to provide axioms $time(A(\vec{x}, \tau)) = \tau$ to extract the timestamp $\tau$ of any primitive action $A(\vec{x}, \tau)$ and the function $start(do(a, s)) = time(a)$ which returns a situation's start time. The initial time $start(S_0)$ may be defined in the BAT. Using these, $transAtPr$ can ensure monotonicity of time:

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

When a *stochastic action* $\beta$ is executed, the idea is that nature randomly picks a primitive outcome action $\alpha$. The axiomatizer is supposed to provide two macros $Choice(\beta, \alpha)$ and $prob_0(\beta, \alpha, s) = p$ as in (Reiter 2001). The former denotes that $\alpha$ is a feasible outcome action of $\beta$, the latter returns the probability of nature actually choosing $\alpha$ 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:

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

[5] For those familiar with ConGolog, this is because we do not consider synchronized if-then-else and while-loops.

### 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. $value$ inspects the tree of situations induced by stochastic actions 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:

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

The $\max$ expression maximizes over all possible decompositions $\gamma; \delta'$:

$$\begin{aligned} &\max_{\{(\gamma,\delta')|Next(\delta,\gamma,\delta')\}} f(\gamma, \delta') = v \stackrel{\text{def}}{=} \\ &\quad \exists \gamma, \delta' \,.\, Next(\delta, \gamma, \delta') \wedge v = f(\gamma, \delta') \wedge \\ &\quad (\forall \gamma', \delta'')(Next(\delta, \gamma', \delta'') \supset v \geq f(\gamma', \delta'')). \end{aligned}$$

$f(\gamma, \delta')$ stands for the sum over all situations reached by $transAtPr$. For an axiomatization of the sum we refer to (Bacchus, Halpern, and Levesque 1999).

### 4.4 Transition Semantics

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

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

The function is consistent, i.e., $transPr(\delta, s, l, \delta', s')$ 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 $\beta$ is executed, this is reflected in $s' = do(\alpha, s)$ for some primitive outcome action $\alpha$ and the only cause of $\alpha$ is $\beta$ 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 $\phi$ which holds in the world at time $\tau$ according to the sensors (e.g., $\phi$ might tell us the position of each car at time $\tau$). For each of the, say, $n$ vehicles, we choose a $\delta_i$ from the pre-defined programs. $\delta_i$ serves as hypothetical explanation for the $i$th 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 $\delta$ 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

$$Poss(\mathsf{match}(\phi, \tau), s) \equiv \phi[s, \tau]$$

which asserts that the observed formula $\phi$ actually holds in the model at time $\tau$. Hence, an executed match action represents an explained observation. Note that although match and waitFor have equivalent semantics, they are different actions to capture their different intentions.

Plan recognition can be carried out online roughly by repeating the following two steps:

(1) If a new observation is present, merge the respective match action into the remaining 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

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

While being greedy is not always optimal, this heuristic allows us to do plan recognition online. Since the interpreter can execute up to $l$ match actions during its look-ahead, nondeterminism is resolved optimally modulo look-ahead $l$ as long as the program contains at least $l$ match actions. Therefore, a more precise formulation of (2) is:

(2) If the remaining program contains at least $l$ match actions, execute the next step of the hypothesis program.

We now detail steps (1) and (2). Let $\delta$ be the hypothesis. The initial plan recognition state is $\{(\delta, S_0, 1)\}$ because, as nothing of $\delta$ has been executed yet, it may be a perfect hypothesis. As time goes by, $\delta$ 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 $\phi$ is observed at time $\tau$ we replace all configurations $(\delta, s, p)$ with new configurations $(\delta \parallel \mathsf{match}(\phi, \tau), s, p)$. When the number of match actions in $\delta$ is at least $l$, we are safe to update the configuration by triggering the next transition. Thus, upon matching the observation $\phi$ at time $\tau$, a state $\mathcal{S}_k$ of the plan recognition evolves as follows:

$$\begin{aligned} \mathcal{S}_{k+1} = \{&(\delta', s', p') \mid (\delta, s, p) \in \mathcal{S}_k \text{ and} \\ &\delta \text{ contains at least } l-1 \text{ match actions and} \\ &\mathcal{D} \cup \mathcal{C} \models p' > 0 \wedge \\ &\quad p \cdot transPr(\delta \parallel \mathsf{match}(\phi, \tau), s, l, \delta', s') = p'\} \\ \cup \{&(\delta \parallel \mathsf{match}(\phi, \tau), s, p) \mid (\delta, s, p) \in \mathcal{S}_k \text{ and} \\ &\delta \text{ contains less than } l-1 \text{ match actions}\} \end{aligned}$$

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 $U_k \subseteq \mathcal{S}_k$ be a set of configurations $(\delta, s, p)$ that stem from *one* of the optimal ways to resolve nondeterminism. Then the confidence of $U_k$ being an explanation so far is

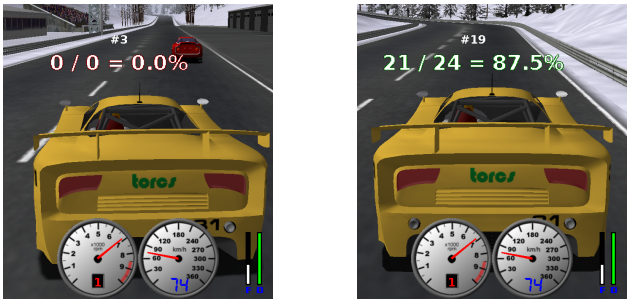
$$\sum_{(\delta,s,p)\in U_k} p \cdot \frac{r(s)}{r(s) + m(\delta)}$$

where $m(\delta)$ is the number of match actions that occur in the program $\delta$. This weighs the probability of reaching the configuration $(\delta, s, p)$ by the ratio of explained observations $r(s)$ in the total number of observations $r(s) + m(\delta)$. Since there are generally multiple $U_k$, the confidence of the whole hypothesis is $\max_{U_k} \sum_{(\delta,s,p)\in U_k} p \cdot \frac{r(s)}{r(s)+m(\delta)}$.

## 6 Classifying Driving Maneuvers

We have implemented a prGolog interpreter and the online plan recognition procedure in ECLiPSe-CLP,[6] a Prolog dialect. We evaluated the system with a driving simulation, TORCS,[7] to recognize driving maneuvers. Our car model is implemented in terms of stochastic actions such as setVeloc and setYaw and continuous fluents like $x$ and $y$ which depend on 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.[8] The interpreter's look-ahead to resolve nondeterministic choice points varies between two and three.

For stochastic actions we use a Monte-Carlo-simulation-like sampling to further increase the performance. That is, for a given program and a set of observations we would normally need to compute the transition probability as the weighted sum of the probability of all outcomes along all possible execution paths of the program. The number of combinations gets very large with increasing number of stochastic actions in the program. So instead, for every program and set of observations to explain, we spawn a number of interpreters and sample the outcomes of stochastic actions appearing in the program according to the outcomes' probability distribution. This way, we can approximate the real transition probability with arbitrary accuracy with increasing number of samples, i.e., increasing number of interpreters. In our evaluation we always used 24 samples.




(a) (b)

Figure 3: Plan recognition output of passing maneuver.

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

$$\phi = \bigwedge_i x(V_i) - \Delta x(V_i) \le X_i \le x(V_i) + \Delta x(V_i) \wedge$$
$$y(V_i) - \Delta y(V_i) \le Y_i \le y(V_i) + \Delta y(V_i).$$

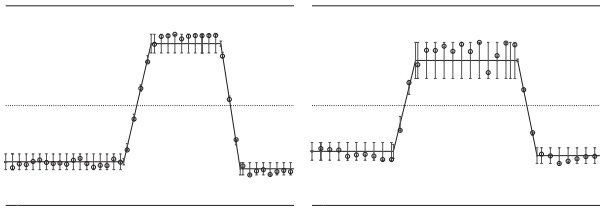
Thus, the plan recognition system needs to search for possible executions of the candidate programs that match the observed car positions. If a smaller tolerance is good enough to match the observations, the confidence in the candidate program being an explanation for the observation is higher, because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

Figure 3 shows how plan recognition results are displayed in the driving simulation. TORCS and the plan recognition system run on two distinct computers which exchange observations and plan recognition results via network. A small coordination program spawns child processes with embedded ECLiPSe-CLP environments which perform the sampling. The plan recognition server has an Intel Core i7 CPU so that six sampling processes run per CPU core. The results are reported back to TORCS and displayed at runtime. 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. A more recent prototype of our system

[6] http://www.eclipseclp.org/
[7] http://torcs.sourceforge.net/
[8] http://www.coin-or.org/

```
do([waitFor(behind(v, w), 7.57),     setYaw(w, 0.00, 0.27, 7.57),
    setVeloc(w, 15.09, 13.22, 7.57), setYaw(v, 0.00, 0.48, 7.57),
    match(..., 8.09),               setVeloc(v, 20.80, 3.65, 8.09),
    match(..., 8.59),               match(..., 9.09),
                % some more match actions
    match(..., 18.63),              setYaw(v, 0.10, 0.81, 18.63),
    match(..., 19.13),              match(..., 19.64),
    match(..., 20.15),              match(..., 20.65),
    setYaw(v, 0.00, 1.40, 20.65),   waitFor(behind(w, v), 20.88),
    match(..., 21.16),              setYaw(v, -0.20, 0.12, 21.42),
    match(..., 21.66),              match(..., 22.17),
    setYaw(v, 0.00, 0.64, 22.28),   match(..., 22.68),
                % some more match actions
    match(..., 28.75)], s0)
```

Figure 4: Sample situation term from v passing w.



(a) (b)

Figure 5: Samples of the passing car during two different passing maneuvers on a two-lane road. The $x$-axis is the time, the $y$-axis is the car's $y$-coordinate on the road. Circles stand for the observed $y$-coordinates. The solid line represents the model car's trace and vertical lines the tolerance.

written in Mercury[9] and also using COIN-OR CLP seems to be significantly faster (the computational latency reduces to less than $0.5$ s under comparable circumstances).

### 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.

In our experiment 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 positive confidence: $0.54$ on average with standard deviation $\pm 0.2$. Interestingly test persons using the keyboard had a higher confidence ($0.71 \pm 0.11$) than those using a steering wheel ($0.46 \pm 0.16$). This is plausible: the steering wheel made it hard to drive straight because it was overly sensitive due to calibration issues.

[9] http://www.mercury.csse.unimelb.edu.au/

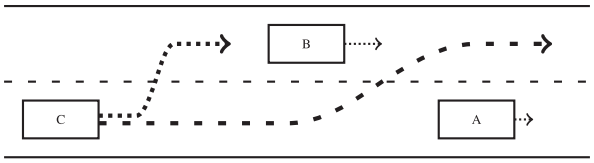



Figure 6: While B passes A, C may choose between two maneuvers.

Figure 4 shows a situation term that resulted from sampling the candidate programs overtake and go_straight. The maneuver begins at time $7.57$ s. Driver w goes straight at about $15$ m/s (on the right lane) and v follows at about $20$ m/s. After $10$ s v swings out, passes w at time $20.9$ s, and finishes the passing maneuver two seconds later. Figure 5 visualizes two samples from two different passing maneuvers. Note that the observed car (represented by circles) oscillates stronger in Figure 5b than in Figure 5a, especially in the fast lane (i.e., above the dashed line). Therefore tolerances in Figure 5b need to be bigger which ultimately levels down the confidence for the hypothesis overtake.

### 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 6. 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, 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. Again, the plan recognition system classified all maneuvers correctly. When C behaved cautiously, this hypothesis was rated $0.3$ on average ($\pm 0.11$) while the aggressive hypothesis was rated with $0.0$. When C drove aggressively, the aggressive program was rated $0.57$ on average ($\pm 0.12$) and the cautious hypothesis was rejected with $0.0$. Hence, the system is able to distinguish correctly between alternative hypotheses.

## 7 Conclusions

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.

However, much more needs to be done to deal with real-world traffic scenarios. For one, we believe that recognition can be improved by moving towards more realistic models of acceleration and the like. The assumption of complete information also needs to be relaxed. Finally, we are interested not only in recognizing which plans are currently being executed but to predict potentially dangerous future situations to assist the driver.

## Acknowledgments

We thank anonymous reviewers for helpful suggestions. The first author is supported by the B-IT Graduate School.

## References

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

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. 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 (AAAI'00)*, 355–362.

Bui, H. H.; Venkatesh, S.; and West, G. 2002. Policy recognition in the abstract hidden markov model. *Journal of Artificial Intelligence Research* 17:451–499.

Charniak, E., and Goldman, R. 1991. A probabilistic model of plan recognition. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, 160–165.

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

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

De Giacomo, G.; Lespérance, Y.; and Muise, C. J. 2011. Agent supervision in situation-determined ConGolog. In *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning, Action and Change (NRAC'11)*, 23–30.

Goultiaeva, A., and Lespérance, Y. 2007. Incremental plan recognition in an agent programming framework. In Geib, C., and Pynadath, D., eds., *Proceedings of the AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR-07)*, 52–59. AAAI Press.

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

Grosskreutz, H., and Lakemeyer, G. 2003b. Probabilistic complex actions in GOLOG. *Fundamenta Informaticae* 57(2-4):167–192.

Kautz, H. A., and Allen, J. F. 1986. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, 32–37.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Liao, L.; Patterson, D. J.; Fox, D.; and Kautz, H. 2007. Learning and inferring transportation routines. *Artificial Intelligence* 171(56):311–331.

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

Pynadath, D. V., and Wellman, M. P. 1995. Accounting for context in plan recognition, with application to traffic monitoring. In Besnard, P., and Hanks, S., eds., *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI'95)*, 472–481. Morgan Kaufmann.

Ramirez, M., and Geffner, H. 2009. Plan recognition as planning. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI'09)*, 1778–1783.

Reiter, R. 1998. Sequential, temporal GOLOG. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 547–556.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.