Towards more realistic logic-based robot controllers in the GOLOG framework

Henrik Grosskreutz, Gerhard Lakemeyer

High-level robot control languages should not only be expressive enough for realistic domains but also support reasoning about actions, in particular, the projection of robot plans, which is useful for the robot when choosing among different courses of action as well as the designer of robot controllers, since projections allow for qualitative simulations. GOLOG, a language based on the situation calculus, was specifically proposed for this purpose. While it comes equipped with a powerful projection mechanism, however, it lacks expressiveness. In particular, it cannot deal with continuous change, event-driven behavior, and probabilistic effects of actions, all of which are important in the domain of mobile robotics. In this paper, we show how these issues can be dealt with in the GOLOG framework by proposing appropriate extensions of the language.

1 Introduction

In the last five years, substantial progress has been made in building mobile robots which can navigate safely in populated areas like office environments, museums, or the like [3,13] .We now have fairly robust solutions to low-level tasks like obstacle avoidance or self-localization so that it is possible to more seriously think about high-level control issues, that is, telling the robot what to do and how to do it. We feel that a high-level robot control language should not only be expressive enough to account for realistic domains in a natural way but also support automated reasoning about the task at hand, in particular, the ability to project the outcome of a given plan or program[1]. The latter is important not only because it is an integral part of intelligent behavior such as rationally choosing among different courses of actions but also for pragmatic reasons. Note that projecting a plan can be thought of as a (qualitative) simulation of how the world evolves when actions are executing, which is quite helpful for debugging purposes. This is especially true for plans with concurrent actions, which arise naturally in robotics applications. Moreover, simulations are in general much faster than actually running tests on the robot.

There have been a number of proposals for high-level control languages such as RPL [18], RAP [6], COLBERT [12], and GOLOG [15]. Among them only RPL and GOLOG allow for plan projection. RPL's projection mechanism called XFRM [18, 19] is problematic, however, because projections rely on using RPL's runtime system, which lacks a formal semantics and which makes predictions implementation dependent. On the other hand, projections in GOLOG, which is based on the situation calculus [17], have a perspicuous declarative semantics. However, in contrast to languages like RPL, GOLOG is by far not expressive enough for realistic robot domains, first successful experiments using GOLOG to control a real museum-tour-guide robot [3] notwithstanding. While there are extensions of GOLOG adding concurrency [8] and time [24], these still do not go far enough. Among the things that are missing we have at least the following:

1. In GOLOG, the world changes in a discrete fashion. However, in the context of mobile robots many changes are best thought of as continuous. For example, while moving, the robot changes its position continuously. The same holds for the battery level or the passage of time. While it may be possible to approximate such changes by discrete approximations, this seems at least unnatural and often adds considerable complexity to the reasoning involved.
2. In the current temporal extension of GOLOG [24], the user has to explicitly supply the time of execution for each action. However, when specifying a robot's task, this seems rarely appropriate and is often infeasible, especially in the context of concurrency. For example, suppose we want to tell the robot to do the following: (1) deliver today's mail to the offices; (2) whenever you pass near Henrik's room say „hello"; (3) whenever the battery level drops dangerously low, interrupt whatever you are currently doing and recharge your batteries. Notice that nowhere do we say explicitly when an action has to be taken. Instead, actions are initiated conditioned on certain events happening like passing a certain office or reaching a low battery level. We call this event-driven behavior.
3. Lastly, actions in GOLOG always have determinate effects, that is, there is no uncertainty about whether or not an action achieves the desired results. In practice, however, uncertainty seems to be ubiquitous, which is in large part due to the shortcomings of today´s robots. Consider, for example, a pickup action. Given a certain characteristic of the gripper and the object to be lifted, we may want to say that the pikkup action succeeds 80% of the time and fails otherwise, which, in its simplest form, may amount to having no effect at all.

In this paper, we will sketch how these shortcomings of GOLOG can be overcome, thus shortening the gap in expressiveness between non-logic-based and logic-based robot control languages. We introduce two extensions of GOLOG called cc-Golog and pGOLOG, respectively. While cc-Golog is concerned with continuous change and event-driven behavior, pGOLOG tackles the issue of actions with uncertain outcome.

[1] We use the terms program and plan interchangeably, following McDermott [18], who takes plans to be programs whose execution can be reasoned about by the agent who executes the program.

The rest of the paper is organized as follows. In the next section we give a very brief introduction to the situation calculus, which is the basic foundation of GOLOG and all its extensions. We then take up cc-Golog and pGOLOG in turn, briefly discuss experimental results and end the paper with some conclusions.

The paper tries to remain largely informal, focussing on the intuitions behind the various extensions of GOLOG rather than details. Those interested in the technical details are referred to [9, 10].

2 The Situation Calculus

One increasingly popular language for representing and reasoning about the effects of actions is the situation calculus [17]. We will only go over the language briefly here: all terms in the language are of sort ordinary objects, actions, situations, or reals[2]. There is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a,s)$ denotes the successor situation of s resulting from performing action a in s; relations and functions whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate and function symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a,s)$ used to state that action a is executable in situation s.

Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [16]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a,s)$
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a,s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [22]. Here is a simple example of a successor state axiom for the fluent $Broken(x,s)$:[3]

$$Poss(a,s) \supset Broken(x, do(a,s)) \equiv$$
$$a = drop(x) \wedge Fragile(x) \vee$$
$$Broken(x,s) \wedge a \neq repair(x)$$

  In other words, provided action a is possible, the object x will be broken after doing a just in case either a is a drop action and x is fragile or x was already broken and a is not a repair action.
- Domain closure and unique names axioms for the primitive actions, as well as unique names axioms for situations.

3 Continuous Change and Time

In order to model continuous processes like a robot's movement along a hallway we begin by adding continuous change and time directly to the ontology of the situation calculus.

As shown in [21, 23], adding time is easy. We simply add a new sort *time* ranging over the reals and, in order to connect situations and time, a special unary functional fluent *start* with the understanding that $start(s)$ denotes the time when situation s begins. We will see later how *start* obtains its values and, in particular, how the passage of time is modeled.

A fundamental assumption of the situation calculus is that fluents have a fixed value at every given situation. In order to see that this assumption still allows us to model continuous change, let us consider the example of a mobile robot moving along a straight line in a 1-dimensional world, that is, the robot's location at any given time is simply a real number. There are two types of actions the robot can perform, $startGo(v)$, which initiates moving the robot with speed v, and *endGo* which stops the movement of the robot. Let us denote the robot's location by the fluent *robotLoc*. What should the value of *robotLoc* be after executing *startGo* in situation s? Certainly it cannot be a fixed real value, since the position should change over time as long as the robot moves. In fact, the location of the robot at any time after $startGo(v)$ (and before the robot changes its velocity) can be characterized (in a somewhat idealized fashion) by the function $x + v \times (t - t_0)$, where x is the starting position and t_0 the starting time. The solution is then to take this *function of time* to be the value of *robotLoc*. We call functional fluents whose values are continuous functions *continuous fluents*.

The idea of continuous fluents is not new and has been investigated in various ways as in [7, 20, 21, 25, 26]. Here we essentially follow Pinto [21] and only illustrate the basic principles by way of example.

For our 1-dimensional robot, we introduce two kinds of functions of time, constant functions, denoted by $constant(x)$ and the special linear functions introduced above, which we denote as $linear(x, v, t_0)$. In order to specify what values these functions take on at any particular time t we use the following axioms:

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

Before considering how these functions of time allow us to specify the successor state axiom of *robotLoc*, let us turn to the issue of modeling the passage of time during a course of actions. Motivated by the treatment of time in robot control languages like RPL, RAP, or COLBERT, we introduce a new type of primitive action $waitFor(\theta)$. The intuition is as follows. Normally, every action happens immediately, that is, the starting time of the situation after doing a in s is the same as the starting time of s. The only exception is $waitFor(\theta)$: whenever this action occurs, the starting time of the resulting situation is advanced to the earliest time in the future when θ becomes true. θ is assumed to be a formula involving functions of time with the situation argument suppressed such as $(robotLoc \geq 1000)$. For any situation s and time t such an expression is „evaluated" by replacing *robotLoc* by $val(robotLoc(s), t)$[4]. Note that choosing the earliest time when θ is true has the effect that actions always happen as soon as possible. One may object that requiring that two actions other than *waitFor* must happen at the same time is unrealistic. However, in robotics applications, actions often involve little more than sending messages in order to initiate or terminate processes so that the actual duration of such actions is negligible. Moreover, if two action cannot happen at the same time, they can always be separated explicitly using *waitFor*.

[2] While the reals are not normally part of the situation calculus, we need them to represent time, continuous change, and probabilities. For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

[3] Throughout, free variables are assumed to be implicitly universally quantified.

[4] Note that there may not always be a least time point where θ is true, for example in the case of $(robotLoc > 1000)$. We deal with this problem be requiring that a *waitFor*-action is possible only if such al least time point exists and leave it to the user to carefully chosse appropriate θ.

With the idea of time advancing only through *waitFor*, we can now specify how the fluent *start* changes its value when an action occurs: the starting time of a situation changes only as a result of a $waitFor(\theta)$, in which case it advances to the earliest time in the future when θ holds. This can easily be expressed in the form of a successor state axiom for *start* which we omit here.

Let us now consider the successor state axiom for the continuous fluent *robotLoc*:

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

In other words, when an action is performed *robotLoc* is assigned either the function $linear(x, v, t_0)$ if the robot starts moving with velocity v and x is the location of the robot at situation s, or it is assigned $constant(x)$ if the robot stops, or it remains the same as in s.

To illustrate what can be derived in this extended situation calculus, let us assume that the robot initially rests at position 0, that is, $(robotLoc(S_0) = constant(0)$. Let us assume the robot starts moving at speed 50 (cm/s) and then waits until it reaches location 1000 (cm), at which point it stops. The resulting situation is $s_1 = do(endGo, do(waitFor(robotLoc = 1000), do(startGo(50), S_0)))$. Then, assuming an appropriate axiomatization of our extended situation calculus, it is possible to show that

$$start(s_1) = 20 \wedge robotLoc(s_1) = constant(1000)$$

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

4 cc-Golog

cc-Golog, a derivative of ConGolog [4], which is a concurrent version of GOLOG, is a formalism for specifying complex actions and how these are mapped to sequences of atomic actions assuming a description of the initial state of the world, action precondition axioms and successor state axioms for each fluent. Complex programs are defined using control structures familiar from conventional programming language such as sequence, while-loops and recursive procedures. In addition, parallel actions are introduced with a conventional interleaving semantics.

$\alpha / waitFor(\theta)$	primitive action
$\phi?$	test action
$seq(\sigma_1, \sigma_2)$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional
$while(\phi, \sigma)$	loop
$withPol(\sigma_1, \sigma_2)$	prioritized execution until σ_2 ends
$tryAll(\sigma_1, \sigma_2)$	concurrent exec. until any σ_i ends
$withCtrl(\phi, \sigma)$	conditioned execution of σ
proc $\beta(x)$ σ	procedure definition

Given the space limitation, we cannot present the formal semantics of cc-Golog in detail. Instead, we will only sketch the intuition behind them (details can be found in [9]). The essential difference between cc-Golog and ordinary programming languages is that cc-Golog's semantics is not defined by specifying which machine instruction are to be performed by the interpreter, but instead to what sequence of atomic situation calculus actions a cc-Golog plan is to be mapped.

As a consequence, the primitive instructions of cc-Golog consist of atomic situation calculus actions (as in GOLOG). Note that a *waitFor* is just an atomic action whose only effect is to make time advance. Besides atomic actions, there is another class of primitive cc-Golog instructions: tests of the form $\phi?$. Here, ϕ stands for a situation calculus formula, for example $isOpen(door6213)$. The intuition behind $\phi?$ is to block if ϕ is false, and else continue with execution. Note that a fundamental difference between conventional programming languages and GOLOG is that a GOLOG interpreter is able to reason about the state of the world (like the state of a door). Reasoning is performed by *regression* [22], a special form of deductive inference, which is quite efficient when the initial description of the world is restricted to a collection of literals.

By means of the other control structures, more complex programs can be composed. The semantics of *seq*, *if* and *while* correspond to their intuitive meaning. *tryAll* and *withPol* specificy that two programs are to be executed concurrently. Intuitively, $tryAll(\sigma_1, \sigma_2)$ starts executing both σ_1 and σ_2; the parallel execution of *tryAll* stops as soon as *one* of σ_1 and σ_2 stop. As for $withPol(\sigma_1, \sigma_2)$, the idea is that a low priority plan σ_2 is executed, which is interrupted whenever the program σ_1, which is called a *policy*, is able to execute. The execution of the whole *withPol* construct ends as soon as σ_2 ends. The possible interleavings resulting from concurrent execution of several programs are ultimately constrained such that actions which can be executed earlier are always preferred. That is, a $waitFor(\theta)$ can only be executed if no concurrent branch of the actual program can execute an earlier action, restoring the original idea that actions should happen as early as possible. Finally, $withCtrl(\phi, \sigma)$ is actually not a primitive instruction but a macro defined in terms of other instructions. Its intended meaning is that it executes σ as long as ϕ is true, gets blocked if ϕ becomes *false* and continues execution if ϕ becomes true again.

Policies offer a natural way to realize event-driven behavior, especially, as the following example illustrates, if it makes use of *waitFor* instructions. Here, we turn back to the introductory example of a robot that is to (1) deliver mail to the offices; (2) say „hello" whenever it passes near Henrik's room; (3) interrupt its actual course of action whenever the battery level drops below 46 Volt and recharge its batteries. This task can be specified through the following cc-Golog plan, where $loop(\sigma)$ is a shorthand for $while(true, \sigma)$.

$$\begin{aligned} &withPol(loop(waitFor(battLevel \leq 46, \\ &\quad seq(grabWhls, chargeBatteries, releaseWhls))), \\ &\quad withPol(loop(waitFor(nearDoor6213, \\ &\qquad seq(say(hello), waitFor(nearDoor6213)))) \\ &\quad withCtrl(wheels, deliverMail))) \end{aligned}$$

In this program, the outermost policy is waiting until the battery level drops to 46. At this point, *grabWheels*, an atomic action whose effect is to set the fluent *wheels* to *false* is immediately executed. This has the effect that the execution of the program *deliverMail* is blocked. It is only after the complete execution of *chargeBatteries* that *wheels* gets released so that *deliverMail* may resume execution (if, while driving to the battery docking station, the robot passes by *Door6213*, it would still say „hello").

We do not go into the details of *deliverMail* except to note that in order to move the robot continuously toward its various goal locations, *deliverMail* will make use of $startGo(x,y)$, $waitFor(atDestination)$ and *stop* actions. Here, $startGo(x,y)$ is a two-dimensional variant of the action $startGo(vel)$ discussed in section *continuous change*, with the additional effect of continuously reduce *battLevel*.

As the example illustrates, the new action *waitFor* together with the notion of concurrent, prioritized execution of policies turns out to be very helpful when it comes to specifying robust robot plans. As mentioned earlier, the concept of an instruction whose effect is to wait until a condition becomes true is common in special non-logic-based robot programming languages such as RPL [18], RAP [6], or COLBERT [12] (the same is true for the concepts of concurrency and priorities). We believe that this is due to the fact that in real robot applications it is typical that the robot is to execute a primary task, like the mail delivery in our example, and at the same time has to monitor and react to continuously changing properties of the domain like the voltage level of the robot's batteries.

Another important feature is the ability of a policy to block the execution of the primary task through the *withCtrl*-instruction. In our example, the battery-monitoring policy must be able to wait for a continuous condition both in blocking and in non-blocking mode: it must not block the primary task while it is waiting until the battery level falls below 46 Volt; but thereafter, it must block the primary task while it guides the robot to the battery loading station and waits until the battery level climbs back to a reasonable level.

5 Probabilistic Projection

Another important feature of real robot environments is the inherent uncertainty in what the world is like and the outcome of many of a robot's actions, due to the fact that robot hard- and software is imperfect and error-prone. For example, if a robot tries to pickup a cup, many different outcomes are possible: the robot may completely miss the cup, the cup may drop on the floor, the robot may push adjacent objects or might even break the cup or an adjacent object.

For several reasons, such robot actions are often best thought of as low-level processes with uncertain, probabilistic outcome. For one, we might want to model, for example, that the pickup succeed perfectly 80% of the time and has some other possible outcomes with lower probability. Second, it is convenient to describe the pickup action as a complex process (as opposed to a primitive situation calculus action). Indeed, the pickup action is not atomic. It may result in the cup being held, it might additionally break an adjacent object if one exists or the like.

If we adopt the point of view that robot actions are better seen as complex low-level processes, a high-level robot plan can be seen as a description of a task which combines such low-level processes in an appropriate way. In order to evaluate and choose an appropriate high-level robot plan, we wish to project the effects of the execution of such a plan. To do that, we need to explicitly model the behavior of the processes. As their outcomes are probabilistic in nature, we will arrive at a notion of *probabilistic projection*.

To attack this problem, we first model the low-level processes by means of procedures in a probabilistic action language, which we call pGOLOG. In a nutshell, pGOLOG is the deterministic fragment of GOLOG augmented with a new instruction, *prob*, which allows us to express that a program is executed only with a certain probability. Intuitively, the execution of $prob(p, \sigma_1, \sigma_2)$ results in the execution of σ_1, resp. σ_2 with probability p resp. $1 - p$.[5] Given a faithful characterization of the low-level processes in terms of pGOLOG procedures, we can then project the effect of the activation of these processes using their corresponding pGOLOG models. We will soon discuss the differences between probabilistic and non-probabilistic projections, but first illustrate how the above example of a low-level pickup process can be specified as a pGOLOG program.

$$
\begin{array}{l}
proc(pickup(Cup) = \\
\quad prob(0.8, perfectPickup(Cup), \\
\quad\quad seq(perfectPickup(Cup), drop(Cup)), \\
\quad if(\exists obj.closeTo(obj, Cup) \wedge fragile(obj), \\
\quad\quad prob(0.5, breakCloseObj, nil)))
\end{array}
$$

This pGOLOG program models that with probability 80% the pickup process will result in a flawless pickup of cup. Else, it will lose the cup after lifting it, and with probability 50% will additionally break a fragile adjacent object if one exists (note that the total probability is thus 10%).

In many typical scenarios, there is also uncertainty about the initial situation. To take this into account, we opt for a probabilistic characterization of an agent's epistemic state. More specifically, we characterize an epistemic state by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities. We thereby follow [1], who introduce a binary functional fluent $p(s', s)$ which can be read as „in situation s, the agent thinks that s' is possible with probability $p(s', s)$." All likelihoods must be nonnegative and situations considered impossible will be given likelihood 0. To keep things simple, we additionally require that the likelihood of all situations considered possible in S_0 sum to 1.

Now we come back to the task of probabilistic projection. Unlike in the non-probabilistic case discussed above, the generation of a projection of a plan doesn't mean any longer that a single unique execution scenario, *the* execution trace of the plan is to be generated. Instead, the execution of a plan can result in *many* different execution traces, because the activation of the low-level processes may result in different outcomes. Our goal is then to assess the degree of belief in sentences like the goal $\neg\exists cup.broken(cup)$ after the execution of a plan. To do so, all possible execution traces have to be considered.

To determine the probability that a sentence ϕ holds after the execution of a plan σ, we determine every possible execution trace of the plan and the activated low-level processes wrt each initial situation considered possible. $Bel(\phi[now], s, \sigma)$, the belief that ϕ holds after the execution of plan σ in a situation s is then defined to be the probability of all execution traces s'' of σ (wrt the low-level processes) that fulfill $\phi[now|s''] (= \phi$ with now replaced by $s'')$, starting from a possible initial configuration s'. The execution traces are additionally weighted by the agent's belief in s'.

Summarizing, while during real execution the actual low-level processes get executed, for the task of projection we model the behavior of the low-level processes by means of probabilistic pGOLOG programs. We stress that from the point of view of the execution system, low-level processes are treated as atomic events - the activation of the process. The pGOLOG procedures only serve as models of the effects of the low-level processes that are only needed during the projections of a plan. Indeed, the execution system cannot execute pGOLOG procedures, for one because it has incomplete or uncertain information about the value of the fluents appearing in the pGOLOG program. In the above example, the robot just may not know whether a nearby object is fragile or not. Besides, there is no way how the execution system can, for example, directly break a cup.

A promising property of this framework is that it is easily amenable to Monte-Carlo methods for the estimation of the success probability of a pGOLOG program. In a nutshell, Monte-Carlo simulation can be achieved by pursuing only one of the bran-

[5] We completely gloss over the technical details discussed in [10].

ches of a *prob* instruction depending on the outcome of a random number toss. The appealing property of Monte-Carlo methods is that the number of samples to be considered depends only on the desired precision of the estimate, not on the length of the program, and therefore is not affected by the combinatorial explosion of the number of possible execution traces.

6 Experimental Results

Although the formal definition of cc-Golog and pGOLOG requires second-order logic, it is easy to implement a PROLOG interpreter for cc-Golog, just as in the case of the original ConGolog[6]. In order to deal with the constraints implied by the *waitFor* instruction, we have made use of the ECRC Common Logic Programming System Eclipse 4.2 and its built-in constraint solver library clpr to implement a cc-Golog interpreter (similar to Reiter [24]).

Using this interpreter, we can generate projections of cc-Golog plans like the example mail delivery plan. Compared to earlier work [2] on the projection of RPL programs using the XFRM system [19], the results using cc-Golog are appealing: the cc-Golog implementation is firmly based on a logical specification, while XFRM relies on the procedural semantics of the RPL interpreter. Furthermore, cc-Golog appears much faster: the projection of example plan of [2] took 0.5 seconds in cc-Golog resp. 3.6 seconds in XFRM on the same machine. We believe that cc-Golog owes this somewhat surprising advantage to the fact that it lends itself to a simple implementation with little overhead, while XFRM relies on the rather complex RPL-interpreter involving many thousand lines of Lisp code.

The results provided by the pGOLOG implementation are similar. We compared the performance of our implementation with that of Buridan, a classical probabilistic planner [14], using the „Bomb/Toilet" and „Slippery Gripper" scenarios of [14]. Again, our approach was not only able to compete with Buridan, but outperformed it by an order of magnitude on the same machine.

7 Conclusions

In this paper, we have shown how several shortcomings of GOLOG can be overcome with the aim of using GOLOG for more realistic high-level robot controllers. With cc-Golog we demonstrated how to deal with continuous change and time. A key feature is the use of a new primitive instruction !!!!!!!, which allows us to model event-driven behavior. In pGOLOG, we showed how to incorporate actions with probabilistic effects into the GOLOG-framework and we defined the notion of probabilistic projection.

An issue left open is how to incorporate the two extensions into one coherent language. We hope to report on this in the future and also on more experiments using the new language for high-level controllers on a mobile robot in realistic domains.

In other work related to GOLOG, our group is involved with diagnosing and repairing execution failures [11]. Finally, work is underway to connect a GOLOG-controller with a real-time speech interface [5].

References

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

[2] Beetz, M., and Grosskreutz, H. 1998. Causal models of mobile service robot behavior. In *AIPS'98*.

[3] Burgard, W.; Cremers, A.; Fox, D.; Hähnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 2000. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence* 114(1-2).

[4] de Giacomo, G.; Lesperance, Y.; and Levesque, H. J. 1997. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *IJCAI'97*.

[5] Dylla, F. 2000. Robust, real-time control of an autonomous robot using speech. Master's thesis, Department of Computer Science, RWTH Aachen. in progress (in German).

[6] Firby, J. 1987. An investigation into reactive planning in complex domains. In *Proc. of AAAI-87*, 202–206.

[7] Galton, A. 1990. A critical examination of Allen's theory of action and time. *Artificial Intelligence* 42:159–188.

[8] Giacomo, G. D.; Lesperance, Y.; and Levesque, H. J. 1999. Congolog, a concurrent program-ming language based on the situation calculus: foundations. Technical report, University of Toronto, http://www.cs.toronto.edu/cogrobo/.

[9] Grosskreutz, H., and Lakemeyer, G. 2000a. cc-golog: Towards more realistic logic-based robot controllers. In *AAAI'2000*.

[10] Grosskreutz, H., and Lakemeyer, G. 2000b. Turning high-level plans into robot programs in uncertain domains. In *ECAI'2000*.

[11] Iwan, G. 1999. Explaining what went wrong in dynamic domains. 23rd Annual German Conference on Artificial Intelligence (KI-99), poster presentation.

[12] Konolige, K. 1997. Colbert: A language for reactive control in sapphira. In *KI'97*, volume 1303 of *LNAI*.

[13] Kortenkamp, D.; Bonasso, R.; and Murphy, R. 1998. *AI-based Mobile Robots: Case studies of successful robot sys-tems*. MIT Press.

[14] Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An al-gorithm for probabilistic planning. *Artificial Intelligence* 76:239–286.

[15] Levesque, H. J.; Reiter, R.; Lesprance, Y.; Lin, F.; and Scherl, R. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

[16] Lin, F., and Reiter, R. 1994. State constraints revisited. *Journal of logic and computation* 4(5):655–678.

[17] McCarthy, J. 1963. Situations, actions and causal laws. Technical report, Stanford University. Reprinted 1968 in Semantic Information Processing (M. Minske ed.), MIT Press.

[18] McDermott, D. 1992. Robot planning. *AI Magazine* 13(2):55–79.

[19] McDermott, D. 1994. An algorithm for prob-abilistic, totally-ordered temporal projection. Research Report YALEU/DCS/RR-1014, Yale University, www.cs.yale.edu/AI/Planning/xfrm.html.

[20] Miller, R. 1996. A case study in reasoning about actions and continuous change. In *ECAI'96*.

[21] Pinto, J. 1997. Integrating discrete and continuous change in a logical framework. *Computational Intelligence, 14(1)*.

[22] Reiter, R. 1991. The frame problem in the situation cal-culus: a simple solution (sometimes) and a ccompleteness result for goal regression. In *In Artificial Intelligence and* Mathematic Theory of Computation: Papers in Honor of *John McCarthy*.

[23] Reiter, R. 1996. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. KR'96*, 2–13.

[24] Reiter, R. 1998. Sequential, temporal golog. In *Proc. KR'98*.

[25] Sandewall, E. 1989. Combining logic and differential equations for describing real-world systems. In *KR'89*, 412–420.

[26] Shanahan, M. 1990. Representing continuous change in the event calculus. In *ECAI'90*.

Contact:

Department of Computer Science,
Aachen University of Technology, D-52056 Aachen, Germany,
grosskreutz,gerhard@cs.rwth-aachen.de

Henrik Grosskreutz studierte Informatik an den Universitäten Würzburg, Caen und Bonn. Seit 1998 ist er Stipendiat im Graduiertenkolleg „Informatik und Technik" and der RWTH Aachen.

Gerhard Lakemeyer received his Ph.D. from the University of Toronto and is currently Associate Professor and Head of the Knowledge-Based Systems Group at Aachen University of Technology. He has served on the programme committee of numerous international confer-ences and is a member of the editorial board of the Journal of Artificial Intelligence Research.