

On-Line Execution of cc-Golog Plans

Henrik Grosskreutz and Gerhard Lakemeyer

Department of Computer Science V

Aachen University of Technology

52056 Aachen, Germany

{grosskreutz,gerhard}@cs.rwth-aachen.de

Abstract

Previously, the plan language cc-Golog was introduced for the purpose of specifying event-driven behavior typically found in robot controllers. So far, however, cc-Golog is usable only for projecting the outcome of a plan and it is unclear how to actually execute plans on-line on a robot. In this paper, we provide such an execution model for cc-Golog and, in addition, show how to interleave execution with a new kind of time-bounded projection. Along the way we also demonstrate how a typical robot control architecture where a high-level controller communicates with low-level processes via messages can be directly modelled in cc-Golog.

1 Introduction

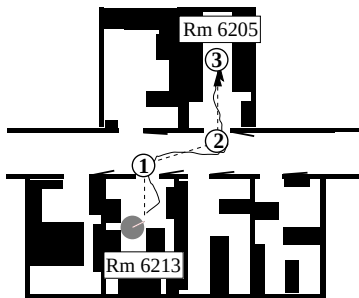


Figure 1: Actual Trajectory and Approximation

Consider a robot which, while engaged in a delivery task, is to move from Room 6213 to Room 6205 (see Figure 1). In a modern robot architecture like XAVIER [Simmons *et al.*, 1997] or RHINO [Burgard *et al.*, 2000], the controller would issue a message to a low-level navigation process telling it the new destination (Rm6205), and the navigation process would then move the robot to its desired location, making use of intermediate goal points like those outside doorways (nodes 1 and 2 in the figure) for robustness.

Besides executing such plans, an intelligent robot controller should be able to reason about its actions, that is, consider, for example, whether it is better to first charge the batteries or to continue the delivery, which would involve projecting the outcome of the delivery task and checking whether the batteries would still have enough power af-

terwards. Moreover, projection is also valuable for the designer of robot controllers to find out whether a program under development is executable or whether it would satisfy certain goals after execution. In [Grosskreutz and Lakemeyer, 2000a], we proposed such a projection mechanism for cc-Golog an extension of GOLOG [Levesque *et al.*, 1997] featuring a model of time, continuous change, and the ability to wait for conditions (like arriving in Rm6205) to become true. While tasks like the above can be modelled quite naturally in cc-Golog, it so far remains unclear how to actually execute a cc-Golog plan or program (in this paper we use plan and program interchangeably). To get a sense of the problem, consider the actions of initiating moving just outside of Room 6213 and waiting to arrive there. In cc-Golog these actions would have the effect of setting the location of the robot to a continuous linear function of time to approximate the robot's trajectory and to advance time to the point when the goal location is reached, respectively. While this seems fine for the purpose of projection, which is sometimes also called off-line execution [de Giacomo and Levesque, 1999], those "effects" are simply inappropriate during actual on-line execution. For one, the robot has no control over the passage of time and, for another, the actual trajectory often follows a function quite different from the idealized approximation (see the curved vs. the straight (dotted) line in Figure 1). What is needed instead, it seems, are frequent sensor readings telling the robot about the current time and location, which should be used instead of the models of how time passes or how the robot moves. A waiting action would then simply reduce to a test whether the goal has been reached. Another minor complication is that actions that are part of the model of a low-level process such as navigation need to be ignored during execution except for an action that activates the process. Finally, while projection in cc-Golog so far is limited to a complete program starting in the initial state, one would often like to project on the fly during execution, similar to the search operator of [de Giacomo and Levesque, 1999]. However, for reasons of efficiency, we would like to go beyond that and allow for a restricted projection of a program, which only searches up to a (temporally) limited horizon. For instance, if the robot is in the middle of a delivery but near the docking station, we want to enable it to find out whether the coming activities would allow it to operate for at least another 5 minutes and, if not, decide to charge the batteries first.

In this paper, we show how all this can be done in cc-Golog. For that we first show how a robot control architec-

ture where a high-level controller communicates with low-level processes via messages can be modelled directly in cc-Golog. The main advantage is that there is a clear separation of the actions of the high-level controller from those of the low-level processes. Then we discuss the changes necessary to use the same cc-Golog program both for on-line execution and projection. Finally, we show how a time-bounded projection mechanism can be defined and interleaved with execution.

The rest of the paper is organized as follows. In Section 2, we briefly review cc-Golog and the situation calculus on which cc-Golog is based. Then we describe how to model the robot control architecture and give an example of a model of a low-level navigation process. In Sections 4 and 5 we discuss the changes needed to allow on-line execution of cc-Golog programs and how on-line execution and projection can be interleaved. We end with a brief summary and discussion of related work.

2 Preliminaries

The Situation Calculus The semantics of cc-Golog is based on an extension of the situation calculus [McCarthy, 1963]. We will only go briefly over the basic situation calculus: all terms in the language are one of the following sorts: ordinary objects, actions, or situations; 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 to s resulting from performing the action a ; relations whose truth values vary from situation to situation are called relational *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; similarly, functions varying across situations are called functional fluents and are denoted analogously; 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 [Levesque et al., 1998]:

- Axioms describing the initial situation, S_0 .
- Action precondition axioms, one for each primitive action a , characterizing $Poss(a, s)$.
- Successor state axioms (SSAs), 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.
- Domain closure and unique name axioms for the actions.
- Foundational, domain independent axioms.

Let us now consider extending this basic situation calculus by time and continuous change according to [Grosskreutz and Lakemeyer, 2000a]. Note that these extensions are intended for the purpose of projections only. In Section 4 we will discuss the necessary changes to allow on-line executions as well. We begin by adding two new sorts real and time.¹ Similar to Pinto and Reiter [Pinto, 1997; Reiter, 1996], we

¹For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations. The sort time ranges over the reals as well.

introduce a special unary functional fluent *start* connecting situations and time. The intuition is that $start(s)$ denotes the time when situation s begins. (We defer the formal definition of *start* until after the discussion of the passage of time.)

As in [Pinto, 1997], continuous change is modelled based on the idea that a fluent like the position of a robot takes as value a *function of time*. For example, in the situation where a robot starts moving, its 2-dimensional location can be characterized (in a somewhat idealized fashion) by a linear function of time, starting at its actual position and moving toward its destination. We call functional fluents whose values are continuous functions *continuous fluents*. In order to represent the value of continuous fluents, we add to the language another sort t-function, whose elements are meant to be functions of time. We assume that there are only finitely many function symbols of type t-function and we require *domain closure* and *unique names axioms* for them, just as in the case of primitive actions. Furthermore, one needs to specify what values these functions have at any particular time t . This is done with the help of the special binary function *val*. The following axiom illustrates the use of *val*, where the values of constant functions and a special kind of linear functions are defined.

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

While $const(x, y)$ always evaluates to the tuple $\langle x, y \rangle$, $linear(x, y, v_x, v_y, t_0)$ is a linear function intended to approximate the movement of a robot in 2-dimensional space (we ignore the details of representing tuples of reals in logic).

Motivated by the fact that during the execution of plans time passes merely when the high-level controller is waiting, the fluent *start* changes its value only as a result of the special primitive action *waitFor*(ϕ). 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*(ϕ): whenever this action occurs, the starting time of the resulting situation is advanced to the earliest time in the future when ϕ becomes true.

The arguments of *waitFor* are restricted to so-called t-forms, which are special formulas involving continuous fluents with the situation argument suppressed (see [Grosskreutz and Lakemeyer, 2000a] for details). An example is $(battLevel \leq 46)$. Given a t-form ϕ , $\phi[s, t]$ denotes ϕ with every continuous fluent evaluated in situation s at time t . For example, $(battLevel \leq 46)[s, t]$ becomes $(val(battLevel(s), t) \leq 46)$. The following axiom defines the least time point after the start of s where ϕ becomes true:

$$\begin{aligned} ltp(\phi, s, t) &\equiv \phi[s, t] \wedge t \geq start(s) \wedge \\ &\forall t'. start(s) \leq t' < t \supset \neg \phi[s, t'] \end{aligned}$$

A *waitFor*(ϕ)-action is possible iff ϕ has a least time point:

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

Finally, the following successor state axiom for *start* captures the intuition that the starting time of a situation changes only as a result of a *waitFor*(ϕ), in which case it advances to the earliest time in the future when ϕ holds.

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

Example To illustrate the use of these definitions, let us model how a robot's location changes as a result of primitive actions. For simplicity, we ignore the robot's orientation and represent its position by the continuous fluent *robotLoc*, which evaluates to a tuple of reals. There are two types of actions that affect the location of the robot: *startGo*($\langle x, y \rangle$), which initiates moving the robot towards position $\langle x, y \rangle$,² and *endGo* which stops the movement of the robot. We assume that *startGo* is only possible if the destination differs from the robot's current location. Then we obtain the following successor state axiom for *robotLoc*.

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

where, $\nu \doteq \sqrt{(x' - x)^2 + (y' - y)^2}$ is a normalizing factor needed in order to ensure that the total 2-dimensional velocity does not exceed 1.

The variables x and y refer to the coordinates of the robot. After *startGo*($\langle x', y' \rangle$), *robotLoc* has as value a linear t-function starting at the current position and moving toward $\langle x', y' \rangle$. After *endGo*, it is *const*($\langle x, y \rangle$). Finally, if a is neither a *startGo* nor a *endGo* action, *robotLoc* remains unchanged.

cc-Golog We now turn to cc-Golog, which offers constructs such as sequences, iteration and procedures to define complex actions. In addition, parallel actions are introduced with a conventional interleaving semantics.

α	primitive action
<i>waitFor</i> (ϕ)	wait until condition ϕ becomes true
$\phi?$	test action
$[\sigma_1, \sigma_2]$	sequence
<i>if</i> (ϕ, σ_1, σ_2)	conditional
<i>while</i> (ϕ, σ)	loop
<i>withPol</i> (σ_1, σ_2)	prioritized execution until σ_2 ends ³
<i>proc</i> $\beta(x)\sigma$	procedure definition

Using cc-Golog, it is possible to specify event-driven behavior quite naturally. For example, the following program specifies that the robot is to travel to Room 6205 and say “In Hallway” if it ever enters the hallway on its way.

prg1 $\doteq withPol([waitFor(inHallway), say("In Hallway")], [send(destRm, 6205), reg(reached) = 6205?])$

Here the *send* action indicates that the robot assigns the room number to a register, which leads to an activation of the navigation process to head to that room, and then blocks execution until the register *reached* signals that the navigation process has completed successfully. The details of this architecture are found in Section 3.

The semantics of cc-Golog is specified similar to that of ConGolog [Giacomo *et al.*, 2000]. Its main ingredients are a predicate *Trans*(σ, s, δ, s'), which defines single steps of

computation, and another predicate *Final*(σ, s). *Final* specifies which configurations (σ, s) are final, meaning that the computation can be considered completed. This is the case, roughly, when the remaining program is *nil*, but not if there is still a primitive action or test action to be executed. We leave out the details for reasons of space.

Intuitively, the predicate *Trans*(σ, s, δ, s') associates with a given program σ and situation s a new situation s' that results from executing a primitive action in s , and a new program δ that represents what remains of σ after having performed that action. For space reasons, we only list a few of the axioms for *Trans*.

$$\begin{aligned} Trans(\alpha, s, \delta, s') &\equiv Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s) \\ Trans(nil, s, \delta, s') &\equiv FALSE \\ Trans(\phi?, s, \delta, s') &\equiv \phi[s] \wedge \delta = nil \wedge s' = s^4 \\ Trans(seq(\sigma_1, \sigma_2), s, \delta, s') &\equiv \\ &Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s') \vee \\ &\exists \delta'. Trans(\sigma_1, s, \delta', s') \wedge \delta = seq(\delta', \sigma_2) \\ Trans(withPol(\sigma_1, \sigma_2), s, \delta, s') &\equiv \neg Final(\sigma_2, s) \wedge \\ &\exists \delta_1. Trans(\sigma_1, s, \delta_1, s') \wedge \delta = withPol(\delta_1, \sigma_2) \wedge \\ &\forall \delta_2, s_2. Trans(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)] \\ &\vee \exists \delta_2. Trans(\sigma_2, s, \delta_2, s') \wedge \delta = withPol(\sigma_1, \delta_2) \wedge \\ &\forall \delta_1, s_1. Trans(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1)] \end{aligned}$$

A transition of a primitive action requires it to be possible in the current situation, leaving nothing (*nil*) to be done afterwards. The execution of σ_2 with policy σ_1 means that one action of one of the programs is performed, whereby actions which can be executed earlier are always preferred. If both σ_1 and σ_2 are about to execute an action at the same time, the policy σ_1 takes precedence.⁵

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

$$\begin{aligned} Do(\delta, s, s') &\equiv \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s') \\ Trans^*(\delta, s, \delta', s') &\equiv \forall T[... \supset T(\delta, s, \delta', s')] \end{aligned}$$

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

$$\begin{aligned} T(\delta, s, \delta, s) \\ Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s') \end{aligned}$$

Given a program δ , proving that δ is executable in the initial situation then amounts to proving $\Sigma \models \exists s Do(\delta, s_0, s)$, where Σ consists of the above axioms for cc-Golog together with a basic action theory in the situation calculus.

3 A Robot Control Architecture

As indicated in the introduction, in modern robot architectures like XAVIER [Simmons *et al.*, 1997] and RHINO [Burgard *et al.*, 2000], the high-level controller does not directly operate the robot's physical sensors and effectors. Instead, it

⁴Here, ϕ stands for a situation calculus formula with all situation arguments suppressed. $\phi[s]$ will denote the formula obtained by restoring situation variable s to all fluents appearing in ϕ .

⁵Note that *Trans* requires a reification of formulas and programs in the logical language. See [Giacomo *et al.*, 2000] for how this can be done.

activates and deactivates specialized processes like a navigation process, to which we will refer as *low-level processes*. The job of the high-level controller is then to combine the activation and deactivation of these routines in a way to fulfill the overall goal. It turns out that cc-Golog allows a logical reconstruction of this type of architecture in a fairly natural way. The overall architecture is illustrated in Figure 2.



Figure 2: Robot Control Architecture

The communication between the low-level processes and the high-level plan is achieved through the special fluent *reg*. The high-level interpreter can affect the value of *reg* by means of the special action *send(id, val)* which assigns *reg(id)* the value *val*. The intuition is that in order to activate a low-level process, the high-level controller executes a *send* action. For example, the execution of *send(destRm, 6205)* would tell the navigation process to start moving toward Room 6205.

On the other hand, the low-level processes can provide the high-level controller with information by means of the (exogenous) action *reply(id, val)*. For example, in order to inform the high-level controller that it has reached its destination, the low-level process would cause an exogenous *reply(reached, 6205)* action. The following successor state axiom specifies how *reg* is affected by *send* and *reply*.

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

3.1 Modeling Low-level Processes as cc-Golog Procedures

To describe complex low-level processes like a navigation process, we model low-level processes as cc-Golog *procedures*.⁶ Given a faithful characterization of the low-level processes in terms of cc-Golog procedures, we can then *project* the effect of the activation of these processes using their corresponding cc-Golog models. We stress that these procedures are not meant to be executed, but rather represent a model of the effects of the corresponding low-level process.

As an example, let us consider a low-level navigation process. It is activated through a *send(destRm, R)* action, which assigns *reg(destRm)* the value *R* and tells the process to travel to room *R*. The process causes the robot to move towards *R* until the destination is reached, where it informs the high-level controller of the arrival by means of an *reply(reached, R)* action.

We model this behavior by the cc-Golog procedure *navProc*. *navProc* makes use of the following functions: *exitOf(R)*, *inside(R)* and *currentRoom*. *exitOf* maps a room name *R* to a location near the exit of room *R*, and *inside* to location within *R*. *currentRoom* is a function whose value in *s* is the name of the room the robot is in at the beginning of *s*. We write *gotoLoc(d)* as an abbreviation for *[startGo(d), waitFor(near(d))]* and *loop(σ)* for *while(TRUE, σ)*.

⁶This is similar to [Grosskreutz and Lakemeyer, 2000b], where a probabilistic programming language is used to model processes with uncertain outcome.

```
navProc ≡ loop([reg(destRm) ≠ currentRoom?,
               if(currentRoom ≠ HALLWAY,
                  gotoLoc(exitOf(currentRoom))),
               gotoLoc(exitOf(reg(destRm))),
               gotoLoc(inside(reg(destRm))), endGo,
               reply(reached, reg(destRm))])
```

Initially, the navigation process is blocked until *reg(destRm)* is assigned a room name different from the one the robot is actually in. Then *navProc* executes a sequence of *startGo* actions, approximating the trajectory towards the destination room by a polyline with an edge near every door the robot has to travel through (see Figure 1).

3.2 Projection

We will now describe how to project a cc-Golog plan, taking into account the cc-Golog model of the low-level processes. Let *s* be a situation, *ll_{model}* a model of the low-level processes,⁷ and *σ* a cc-Golog program. A projection *s'* can then be identified with the situation that results from the concurrent simulation of *σ* and *ll* starting in *s*.

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

Let *AX_{cc}* be the set of foundational axioms of Section 2 together with the domain closure and unique names axioms for t-functions, the axioms required for t-form's, the axioms defining *val*, the precondition axiom for *waitFor*, and axioms stating that the robot is in Room 6213 together with other appropriate descriptions of the environment like a definition of *inHallway* in terms of coordinates on the map. Using *AX_{cc}*, we can project the plan *prg1* from Section 2. We write *do([a₁, ..., a_n], s)* a shorthand for *do(a_n, ..., do(a₁, s) ...)*.

$$\begin{aligned} AX_{cc} \models Proj(S_0, prg1, navProc, s) \equiv \\ s = do([send(destRm, 6205), startGo(l_1) \\ waitFor(near(l_1)), startGo(l_2), \\ waitFor(inHallway), say("In Hallway") \\ waitFor(near(l_2)), startGo(l_3), \\ waitFor(near(l_3)), endGo \\ reply(reached, 6205)], S_0) \end{aligned}$$

The projected execution trace includes a *startGo(l_i)* and *waitFor(near(l_i))* action for every node on the approximating trajectory, where the *l_i* stand for coordinates corresponding to the nodes *i* in Figure 1. Additionally, it includes a *say("In Hallway")* which is executed immediately after the hallway has been reached, which is assumed to happen just after leaving *near(l₁)*. The execution trace ends with *reply(reached, 6205)*, which completes the navigation task.

4 On-Line Execution of cc-Golog plans

Projections such as the above should be understood as a way of assessing whether a program is executable *in principle*. The resulting execution trace is *not* intended as input to the execution mechanism of the robot for several reasons. For one, many of the actions like *startGo(l₁)* only serve to model the navigation process and are not meant to be executed by the high-level controller at all. For another, actions that do belong to the high-level controller like *waitFor(inHallway)* must be interpreted differently during on-line execution. This

⁷In our example, the navigation process is the only low-level process, so we can simply use *navProc*. If there were more relevant low-level processes, *ll_{model}* would amount to the concurrent execution of the individual cc-Golog models.

is because during projection *waitFor* advances time to the least point where the condition is satisfied according to an idealized model of the world (like piece-wise linear trajectories). During actual execution, however, a *waitFor* should intuitively be treated simply as a test, where a condition like being at a certain location is matched against sensor readings reflecting the actual state of affairs.

In order to see how such sensor readings can be obtained, let us briefly look at how actual robots like the RHINO system deal with it. There we find a tight update loop between the low-level system and the high-level controller. This update loop periodically provides the high-level controller with an update of the low-level processes' estimate of continuous properties like the batteries' voltage level or the robot's position (typically several times a second). The period of time between two subsequent updates is so small that for practical purposes the latest update can be regarded as providing not only the correct current time but also accurate values of the continuous fluents at the current time.

Our solution is then to represent the updates by means of a new exogenous action *ccUpdate* and to treat *waitFor*'s simply as special tests during on-line execution. Intuitively, the effect of *ccUpdate* is to set the value of the continuous fluents to the latest estimates of the low-level processes. The actual arguments of *ccUpdate* depend on the continuous fluents that are to be updated. In our example scenario, *ccUpdate* has four arguments: x, y, l and t , where x and y refer to the current position of the robot, l to the current voltage level and t to the current time. To get a sense of the effect of *ccUpdate*, let us consider a new version of the successor state axiom for *robotLoc* of Section 2 suitable for both on-line execution and projection. For simplicity, it is assumed that the actions *startGo* and *endGo* only occur during projection (as part of the model of the low-level navigation process).

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

Note that *ccUpdate* simply "assigns" f the constant function $const(x', y')$, which reflects the idea that, during on-line execution, the linear approximation of the trajectory plays no role and that the actual values $\langle x', y' \rangle$ should be used instead.

Besides updating the value of the continuous fluents, *ccUpdate* is of prime importance because it causes time or, better, the internal clock to advance during execution. Therefore, we need to modify the successor state axiom of the fluent *start*. In order to account for the fact that time advances differently in projections and during on-line execution, we need to explicitly distinguish between the two modes of operation. For that purpose, we introduce a special fluent *online(s)* which can only change its truth value by the special actions *setOnline* and *clipOnline*, respectively. (Specifying the SSA for *online* is easy and left out.) The SSA for *start* then becomes

$$Poss(a, s) \supset [start(do(a, s)) = t \equiv \neg online(s) \wedge \exists \phi. a = waitFor(\phi) \wedge ltp(\phi, s, t) \vee$$

$$online(s) \wedge \exists x_u, y_u, l_u. a = ccUpdate(x_u, y_u, l_u, t) \vee \\ t = start(s) \wedge (\neg online(s) \supset \forall \phi. a \neq waitFor(\phi)) \wedge \\ (online(s) \supset \forall \vec{x}. a \neq ccUpdate(\vec{x}))].$$

In other words, $start(do(a, s))$ is assigned the least time point when the *waitFor*-condition becomes true under off-line execution, or the time value returned by *ccUpdate* under on-line execution, or it remains the same. We remark that we could equivalently write two separate successor state axioms for the on-line and off-line case using the idea of guarded successor state axioms proposed in [Giacomo and Levesque, 1999]. While this would enhance the readability and modularity of the axioms, we do not pursue this idea further here for space reasons.

The precondition axiom for *ccUpdate* ensures that the starting time of legal action sequences is monotonically non-decreasing:

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

A *waitFor*(ϕ) instruction is treated as a special kind of *test* during on-line execution: it succeeds immediately iff the condition ϕ is true *at the beginning* of the actual situation. To ensure this intended behavior, we modify the definition of *Trans* for primitive actions:

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

Finally, we remark that another difference between projection and on-line execution is that during actual execution, whenever the high-level controller executes a *send* action, the interpreter should check whether this signals an activation of a low-level process and, as a side-effect, activate the actual low-level process if necessary.

On-Line Execution The situations which can result from the on-line execution of a plan have a special structure. Basically, they are made up of the sequence of robot actions as defined by the possible transitions (similar to the execution traces defined by $Trans^*$), however with an arbitrary number of exogenous actions between any two robot actions. Formally, the situations s' which can result from the execution of a program σ in situation s can be characterized by the following predicate $Trans^{exo}$:

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

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

$$\begin{aligned} T(\delta, s, \delta, s) \\ Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s') \\ exog(a) \supset T(\delta, s, \delta, do(a, s)), \text{ where} \\ exog(a) \equiv \exists i, n, \vec{x}. a = reply(i, n) \vee a = ccUpdate(\vec{x}) \end{aligned}$$

Note that this differs from the original $Trans^*$ in Section 2 only in the last conjunct, which expresses that an arbitrary number of exogenous actions may occur between any ordinary transition. As an example, let us consider a possible on-line execution trace of *prg1*. *prg1*'s first action is *send(destRm, 6205)*, which results in an activation of the navigation process, which we then assume to provide the high-level controller with *ccUpdate* actions every .25 sec. The execution trace results in situation S_{exec} , where the p_i, q_i stand for appropriate $x - y$ -coordinates along the path of the

robot (we have left out the 3rd argument (voltage level) of *ccUpdate*).

$$S_{exec} \doteq do([send(destRm, 6205), \\ ccUpdate(p_1, q_1, 0.25), ccUpdate(p_2, q_2, 0.5), \\ ccUpdate(p_3, q_3, 0.75), ccUpdate(p_4, q_4, 1.0), \\ say("In Hallway"), ..., ccUpdate(p_{30}, q_{30}, 7.5), \\ reply(reached, 6205)], S_0).$$

It is not hard to see that S_{exec} is a legal on-line execution trace of the introductory cc-Golog program *prg1*, provided the coordinates $\langle p_4, q_4 \rangle$ are the first to satisfy *inHallway* and $\langle p_{30}, q_{30} \rangle$ is near the destination (l_3). Note that *waitFor* no longer occurs in the action sequence since it is now merely a test. Let AX_{ccx} be similar to AX_{cc} but with the new definitions of this section concerning *Trans*, *start*, *ccUpdate* and *robotLoc*. Let Γ be $AX_{ccx} \cup online(S_0)$. Then

$$\Gamma \models \exists \delta. Trans^{exo}(prg1, S_0, \delta, S_{exec}) \wedge Final(\delta, S_{exec})$$

5 Interleaving Projection and On-Line Execution

[de Giacomo and Levesque, 1999] suggest that it is often useful to interleave on-line execution and projection. With our model of time we can take this idea one step further and define projection during on-line execution with the possibility to explicitly put a time-bound on the projection.

5.1 Projection in non-initial Situations

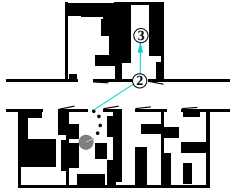


Figure 3: Projection during Execution

So far, we have only considered projection of cc-Golog in the initial situation. To see how things work in non-initial situation, let us again consider our example scenario, where projection makes use of a model of the navigation process. Suppose we are in the situation illustrated in Figure 3, resulting from the on-line execution of *prg1*. The controller begins execution by activating the navigation process through *send(destRm, 6205)*, after which the low-level process provides the high-level controller with a sequence of *ccUpdates* (the *ccUpdates* are indicated as black dots in the figure). The controller is now in situation S_1 with

$$S_1 \doteq do([send(destRm, 6205), \\ ccUpdate(p_1, q_1, 0.25), ccUpdate(p_2, q_2, 0.5), \\ ccUpdate(p_3, q_3, 0.75), ccUpdate(p_4, q_4, 1.0)], S_0).$$

First, let us consider the value of the continuous fluent *robotLoc* in S_1 . Let Γ be defined as before. Then

$$\Gamma \models robotLoc(S_1) = const(p_4, q_4).$$

We can also infer that what remains from *prg1* in S_1 differs from the initial plan (because *send(destRm, 6205)* has already been executed).

$$\Gamma \models \exists \delta. Trans^{exo}(prg1, S_0, \delta, S_1) \wedge \\ \delta = withPol([waitFor(inHallway), say("In Hallway")], \\ [reg(reached) = 6205])?$$

Given the updated value of *robotLoc*, we can now correctly project that the remaining plan will cause the robot to directly travel to its destination. Note the use of *clipOnline*, which forces a switch to projection mode.

$$\Gamma \models \forall \delta, s. Trans^{exo}(prg1, S_0, \delta, S_1) \supset \\ Proj(do(clipOnline, S_1), \delta, navProc, s) \equiv \\ s = do([startGo(l_2), \\ waitFor(inHallway), say("In Hallway"), \\ waitFor(near(l_2)), startGo(l_3), \\ waitFor(near(l_3)), endGo, \\ reply(reached, 6205)], do(clipOnline, S_1))]$$

The reason why projections in non-initial situations of the example scenario are fairly straightforward is that we never have to remember the state the actual navigation process is in. More precisely, for any given position, the cc-Golog model of the navigation process yields an appropriate approximation of the remaining trajectory of the robot (as illustrated in the above projection). While many low-level processes used in mobile robots seem to have this property, we remark that processes where one needs to keep track of their internal state during execution can be dealt with in a way similar to [Grosskreutz, 2000]. The details, however, are outside the scope of this paper.

5.2 Projection Tests

In order to allow for a time-bounded lookahead, we define the predicate $Proj(\phi, t, \sigma, s)$, which is true, roughly, if ϕ holds at time t in the projected execution of a plan σ in situation s . Let ll_{model} refer to an appropriate model of the low-level processes, as before.

$$Proj(\phi, t, \sigma, s) \equiv \\ \exists \sigma^*, s^*. \phi(s^*) \wedge start(s^*) \leq t \wedge \\ Trans^*(withPol(ll_{model}, \sigma), do(clipOnline, s), \sigma^*, s^*) \\ \wedge [\forall \sigma^{**}, s^{**}. Trans^*(\sigma^*, s^*, \sigma^{**}, s^{**}) \supset start(s^{**}) > t \\ \vee Final(\sigma^*, s^*)]$$

Again, we use *clipOnline* to switch to projection mode. The disjunct involving *Final* covers the case where the projected plan ends before time t . Using *Proj* within test conditions, a cc-Golog plan can check whether a possible behavior would lead to certain conditions, and thus deliberate over different possible subplans. In addition, the lookahead can be limited to a certain amount of time t , which seems very useful in order to make quick decisions. A possible application of this test, illustrated by the following program, would be to check if the battery level is going to drop below 46V in the next 300 seconds if the robot comes close to the battery docking station.

$$withPol([waitFor(nearBattDockingStation), \\ if(Proj(deliverMail, 300, battLevel \leq 46), \\ chargeBatteries)], \\ deliverMail)$$

6 Implementation

Although the definition of cc-Golog requires second-order logic, it is easy to implement a PROLOG interpreter for cc-Golog, just as in the case of the original ConGolog.⁸ In order to deal with the constraints implied by the *waitFor* instruction

⁸The subtle differences between the second order axiomatization of ConGolog and a PROLOG implementation are discussed in [Giacomo et al., 2000].

during projection, we have made use of the ECRC Common Logic Programming System Eclipse 4.2 and its built-in constraint solver library `c_lpr` (similar to Reiter [Reiter, 1998]).

As for the on-line execution of `cc-Golog` plans, we have implemented a prototype runtime system that couples the `cc-Golog` interpreter to the `beeSoft` execution system [Burgard *et al.*, 2000]. The link between our PROLOG implementation and the `beeSoft` execution system is based on the High-Level Interface HLI [Hähnel *et al.*, 1998] which provides a uniform PROLOG interface to the low-level processes of `beeSoft`, along with a monitoring component which provides status update about the state of execution of the activated modules. In particular, our runtime-system handles high-level *send* actions, which are mapped to commands to the low-level processes, and periodically generates *ccUpdate* actions reflecting the latest status updates of the low-level processes.

7 Summary and Discussion

In summary, we have extended `cc-Golog` so that it becomes suitable for both projections of plans and their on-line execution. In doing so, we also incorporated a model of a robot architecture typically found on modern mobile robots. Finally, we showed how to interleave on-line execution and a form of time-bounded projection.

As mentioned before, the idea of modeling both projection and on-line execution was first explored in [de Giacomo and Levesque, 1999]. However, the authors consider neither time nor the idea of low-level processes which interact with a high-level controller in complex ways. Most importantly, there is no distinction between the effects on fluents during projection and on-line execution, a distinction we feel is necessary when it comes to modeling essential features of mobile robots such as their location at a given time. [Lesperance and Ng, 2000] have extended de Giacomo and Levesque's ideas in a direction which bears some resemblances to ours. They propose that during projection one also needs to consider a *simulated environment* which, for example, produces exogenous actions to inform the high-level controller that the destination is reached. However, their notion of a simulated environment remains fairly simple since they are not able to model temporally extended processes, and they also do not distinguish between the effects on fluents during projection and on-line execution. We feel that our proposal addresses some of these shortcomings and brings logic-based robot controllers yet another step closer to real world robotics.

The examples in this paper involving navigation tasks of a single robot were chosen mainly because of their simplicity and as such may not be too convincing as to why logic-based robot controllers with built-in projection mechanisms are beneficial or even necessary, especially at run-time. We believe that the advantages of projections will become much more apparent once robots engage in more complex tasks. Consider, for example, a multi-robot delivery scenario where a user makes a request to have a letter delivered by one of the robots. Then in order to determine which robot should deliver the letter, each robot might use projection to determine the cost that would arise if it were to carry out this job, and the task could then be assigned to the robot with a minimal cost estimate. On-the-fly projection becomes even more important when robots need to coordinate their activities. Suppose that two robots agree to meet somewhere at a certain time in the future. Until then they would probably want to carry out

as much of their other duties as possible. Since tasks may not be interruptible at arbitrary times, each robot would be well-advised to check how much of the current task can be completed before heading off to the meeting point. To do so, some form of projection seems necessary. We plan to investigate these and other multi-robot scenarios in the future.

References

- [Burgard *et al.*, 2000] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.
- [de Giacomo and Levesque, 1999] G. de Giacomo and H.J. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.
- [Giacomo and Levesque, 1999] G. De Giacomo and H. Levesque. Projection using regression and sensors. In *Proc. IJCAI'99*, 1999.
- [Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lesperance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.
- [Grosskreutz and Lakemeyer, 2000a] H. Grosskreutz and G. Lakemeyer. `cc-golog`: Towards more realistic logic-based robot controllers. In *AAAI'2000*, 2000.
- [Grosskreutz and Lakemeyer, 2000b] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *ECAI'2000*, 2000.
- [Grosskreutz, 2000] H. Grosskreutz. Probabilistic projection and belief update in the `pgolog` framework. In *Second International Cognitive Robotics Workshop*, 2000.
- [Hähnel *et al.*, 1998] D. Hähnel, W. Burgard, and G. Lakemeyer. Golex - bridging the gap between logic (`golog`) and a real robot. In *Proceedings of the 22st German Conference on Artificial Intelligence (KI 98)*, 1998.
- [Lesperance and Ng, 2000] Y. Lesperance and H.-K. Ng. Integrating planning into reactive high-level robot programs. In *Second International Cognitive Robotics Workshop*, 2000.
- [Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lesprance, Fangzhen Lin, and Richard Scherl. `Golog`: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
- [Levesque *et al.*, 1998] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. URL: <http://www.ep.liu.se/ea/cis/1998/018/>.
- [McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University. Reprinted 1968 in *Semantic Information Processing* (M.Minske ed.), MIT Press, 1963.
- [Pinto, 1997] Javier Pinto. Integrating discrete and continuous change in a logical framework. *Computational Intelligence*, 14(1), 1997.
- [Reiter, 1996] Ray Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. KR'96*, pages 2–13, 1996.
- [Reiter, 1998] R. Reiter. Sequential, temporal `golog`. In *Proc. KR'98*, 1998.
- [Simmons *et al.*, 1997] R.G. Simmons, R. Goodwin, K.Z. Haigh, S. Koenig, J. O'Sullivan, and M.M. Veloso. Xavier: Experience with a layered robot architecture. *ACM SIGART Bulletin Intelligence*, 8 (1–4), 1997.