From Discrete Task Plans to Continuous Trajectories

Ozan Caldiran and **Kadir Haspalamutgil** and **Abdullah Ok** and **Can Palaz**
Esra Erdem and **Volkan Patoglu**
Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul, TURKEY

Abstract

We present a logic-based framework to provide robots with high-level reasoning, such as planning. This framework uses the action description language $\mathcal{C}+$ to represent actions and changes, and the system CCALC to reason about them. In particular, we can represent action domains that involve concurrent actions and additive fluents; based on this description, we can compute shortest plans to a planning problem that involves cost constraints. We show the applicability of this framework on two LEGO MINDSTORMS NXT robots: we compute a discrete task plan (possibly with concurrency) with a cost less than a specified value, and transform this plan into a continuous collision-free trajectory.

Introduction

There have been various studies to close the gap between traditional robotics and cognitive robotics (Levesque and Lakemeyer 2007), by implementing high-level robot control systems based on different families of formalisms for reasoning about actions and change. For instance, (Levesque and Pagnucco 2000) describes a system, LEGOLOG[1], that controls a LEGO MINDSTORMS RIS robot using the high-level control language GOLOG (Levesque et al. 1997) based on the situation calculus (McCarthy 1963; Levesque, Pirri, and Reiter 1998). (Hähnel, Burgard, and Lakemeyer 1998) presents an execution monitoring system for GOLOG and the RHINO control software which operates on RWI B21 and B14 mobile robots. (Ferrein, Fritz, and Lakemeyer 2005) studies coordination of soccer playing robots, using an extension of GOLOG. In the WITAS Unmanned Aerial Vehicle Project[2] temporal action logic (Doherty et al. 1998), features and fluents (Sandewall 1994), and cognitive robotics logic (Sandewall 1998) are used for representing the actions and the events, as a part of a helicopter control system (Doherty et al. 2000). (Shanahan and Witkowski 2000) describes how event calculus (Kowalski and Sergot 1986; Miller and Shanahan 1999) can be used to provide high-level control for a Khepera robot. The agent programming language FLUX (Thielscher 2005), based on the fluent calculus (Thielscher 1998), has also been used to control the execution of some robots.[3] For instance, (Fichtner, Großmann, and Thielscher 2003) presents how FLUX can be used for monitoring the execution of a plan, on a Pioneer 2 mobile robot.

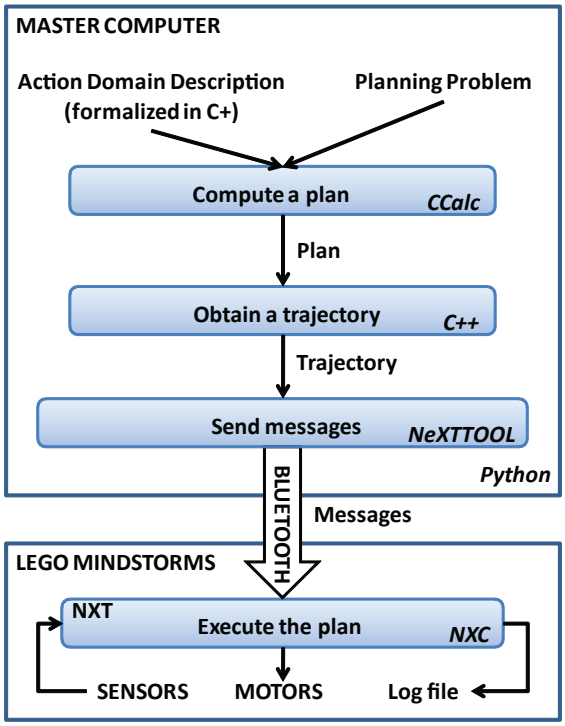

Figure 1: The overall system architecture.

Recently (Caldiran et al. 2009) has presented a formal framework for high level reasoning in the style of cognitive robotics, using the action description language $\mathcal{C}+$ (Giunchiglia and Lifschitz 2004) with the reasoner CCALC[4] on LEGO MINDSTORMS NXT robots. $\mathcal{C}+$ can handle the frame problem (McCarthy and Hayes 1969), the qualification problem (McCarthy 1980), ramifications (Finger 1986), concurrency, numeric-valued fluents (Lee and Lifschitz 2003; Erdem and Gabaldon 2005). LEGO MINDSTORMS NXT is available at a relatively low price and is widely available all over the world compared to more so-

[1] http://www.cs.toronto.edu/cogrobo/Legolog
[2] http://www.ida.liu.se/ext/witas
[3] http://www.fluxagent.org/projects.htm
[4] http://www.cs.utexas.edu/users/tag/cc

phisticated robots. It allows one to build various kinds of robots, and write programs to control them. We continue this line of research by extending this framework to planning problems with cost constraints. In particular, we study planning problems that require two robots to pick up and carry a payload from an initial location to a goal location, on a maze, while avoiding obstacles and that satisfy some cost constraints. The idea is for the robots to automatically generate a plan, and then execute it collaboratively (Fig. 1).

In the rest of the paper, first we describe the overall system shown in Fig. 1. After we describe the particular action domain and the kind of planning problems we are interested in, we formalize them in the language of CCALC. After that, we explain how a plan computed by CCALC is executed by a LEGO MINDSTORMS NXT robot. We conclude with a discussion on the results and the challenges, as well as the future work.

The Overall System Architecture

The overall architecture of our high-level reasoning and control platform is illustrated in Fig. 1.

We start with a description of an action domain in the action description language $\mathcal{C}+$ (Giunchiglia and Lifschitz 2004). The idea is, based on this description, to plan the actions of two LEGO MINDSTORMS NXT robots to achieve a common goal. For that, we use the reasoner CCALC. Given an initial state and goal conditions, CCALC computes a discrete plan to reach a goal state, and displays the complete history (including the state information). From such a history, we extract the continuous trajectories of the robots (including the positions and the orientations of the joints of the robots) using inverse kinematics; these trajectories are obtained from a history automatically with a C++ program. After that, we pass these trajectories to the robots, by means of messages via Bluetooth communication, using the program NeXTTool. All these tasks are automatically performed on a PC using a Python program.

The brain of a LEGO MINDSTORMS NXT robot is NXT—an embedded controller (with an ARM7 microprocessor) capable of processing messages via the Bluetooth communication, and sending signals to three motors. In our work, two motors are used for movements of the robot on a plane; a third motor is used for the rotation of the robot arm. Since gripping would require an additional degree of freedom, a permanent magnet is used as the end-effector; by this way, a payload with metal endpoints can be grabbed by the robots. Several methods and languages exist for programming NXTs. Due to its documentation and relative ease of use, we use the programming language NXC to control the movements of the robots according to the received messages.

Example: Two Robots and a Payload

Consider two robots, and a payload (a long metal stick) on a platform. Suppose that each robot has a magnet at its end-effector so that it can hold the payload only at one end. None of the robots can carry the payload alone; they have to hold the payload at both ends to be able to carry it. Moving in any direction by more than one step increments the cost of a plan by 1 unit. The goal is to place the payload at a specified goal position on the platform with a total cost less than a specified value.

Action Domain Description

We view the platform as a maze. We represent the robots by the constants `r1` and `r2`. We describe the payload by its end points, and denote them by the constants `pl1` and `pl2`.

We characterize each robot by its end-effector, and describe its position by a grid point on the maze. The location `(X,Y)` of a robot `R` is specified by two functional fluents, `xpos(R)=X` and `ypos(R)=Y`. Similarly, the location `(X,Y)` of an end point `P1` of the payload is specified by two fluents, `xpay(P1)=X` and `ypay(P1)=Y`. Movements of a robot `R` in some direction `D` are described by actions of the form `move(R,D)`. Each such action has an attribute that specifies the number of steps to be taken by the robot. We describe the cost of a plan by an additive fluent `cost`.[5]

In the following, suppose that `R` denotes a robot, `P1` and `P2` denote the end points of the payload, `N` and `N1` range over nonnegative integers `1`, ..., `maxN`, and `D` and `D1` range over all directions, `up`, `down`, `right`, `left`. Also suppose that `X1`, `X2`, `Y1`, `Y2` range over nonnegative integers `1`, ..., `maxXY`.

We present the causal laws in the language of CCALC.

Direct effects of actions We describe the effect of a robot's moving right, by the causal laws

```
move(R,right) causes xpos(R)=X2
  if steps(R,right)=N & xpos(R)=X1
  where X2=X1+N & X2 =< maxN.
```

Similarly, we describe the effects of moving in other directions.

Moving in any direction by more than 1 step increments the cost of a plan by 1 unit.

```
move(R,D) increments cost by 1
  if steps(R,D)>1.
```

Ramifications If a robot `R` is at the same location as an end point `P1` of the payload, the end-effector of that robot attracts that end point:

```
caused on(R,P1) if xpos(R)=xpay(P1) &
  ypos(R)=ypay(P1).
```

Then the location of the payload is determined by the locations of the robots:

```
caused xpay(P1)=X1 if on(R,P1) & xpos(R)=X1.
caused ypay(P1)=Y1 if on(R,P1) & ypos(R)=Y1.
```

Preconditions of actions We describe that a robot cannot move in opposite directions by the causal laws

```
nonexecutable move(R,up) & move(R,down).
nonexecutable move(R,left) & move(R,right).
```

[5] An *additive fluent* is a numeric-valued fluent on which the effect of a concurrent action is computed by adding the effects of its primitive actions.

We describe each robot's range of motion, taking into account the Pythagorean Theorem, by the causal laws

```
nonexecutable move(R,D) & move(R,D1)
  if D @< D1 & steps(R,D)=N & steps(R,D1)=N1
  where N*N+N1*N1 > maxN*maxN.
```

The robots can carry the payload only if both of them hold the payload at its end points.

```
nonexecutable move(R,D) if -canCarry & on(R,P1).
```

The conditions under which two robots can carry the payload are described by `canCarry`:

```
caused canCarry
  if on(r1,P1) & on(r2,P2) & P1\=P2
  after on(r1,P1) & on(r2,P2) & P1\=P2.
```

Note that it is required by the causal laws above that the robots wait for one step immediately after they hold the payload at both ends.

Constraints We make sure that the cost of a plan is less than or equal to a specified value `maxCost` by the causal law

```
caused false if cost > maxCost.
```

We describe the constraint that a payload cannot move places unless it is carried by the causal laws

```
caused false if xpay(P1)=X1 & X1\=X2
  after -canCarry & xpay(P1)=X2.
caused false if ypay(P1)=Y1 & Y1\=Y2
  after -canCarry & ypay(P1)=Y2
```

Since CCALC can only deal with integers, we cannot keep track of the exact locations of the payload. (Consider, for instance, moving one end of the horizontally-situated payload up by 2 steps.) Therefore, we allow the payload's length change with a small tolerance for a more flexible motion. Suppose that `linklengthsq` denotes the square of the length of the payload; and `tolerance` denotes the maximum change allowed in the payload's length. The following laws ensure that the payload's length cannot increase/decrease more than `tolerance`:

```
caused false
  if xpay(pl1)=X1 & xpay(pl2)=X2 &
     ypay(pl1)=Y1 & ypay(pl2)=Y2
  where
    (X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1) <
      (linklengthsq-tolerance) ++
    (X2-X1)*(X2-X1)+(Y2-Y1)*(Y2-Y1) >
      (linklengthsq+tolerance).
```

To take care of obstacles on the platform (to prevent collisions), we add the following causal laws:

```
caused false if xpos(R)=X1 & ypos(R)=Y1
  after xpos(R)=X2 & ypos(R)=Y2
  where collision(X1,Y1,X2,Y2) &
    between(X2-maxN,X2+maxN,X1) &
    between(Y2-maxN,Y2+maxN,Y1).
caused false
  if xpay(pl1)=X1 & ypay(pl1)=Y1 &
    xpay(pl2)=X2 & ypay(pl2)=Y2
  where collision(X1,Y1,X2,Y2).
```

Here `collision` is an external function defined in C++, and `between` is an external SWI Prolog function; both are evaluated in SWI Prolog while grounding the causal laws. The first law above prevents the robot end-effectors from moving to a position occupied by an obstacle. The second law ensures that at every state of the world the payload cannot collide with an obstacle.

Task Planning with CCALC

With the action domain description above, CCALC can compute solutions to a given planning problem. For instance, suppose that initially the robots `r1` and `r2` are at `(0,0)` and `(0,5)` respectively, and the end points of the payload are at the same locations. The goal is to move the payload to a location so that its end points are at `(10,0)` and `(10,5)`. This planning problem can be described in the language of CCALC by means of a "query" as follows:

```
:- query
maxstep :: 0..infinity;
0: -canCarry, xpos(r1)=0, ypos(r1)=0,
   xpos(r2)=0, ypos(r2)=5,
   xpay(pl1)=0, ypay(pl1)=0,
   xpay(pl2)=0, ypay(pl2)=5;
maxstep: xpay(pl1)=10, ypay(pl1)=0,
   xpay(pl2)=10, ypay(pl2)=5.
```

CCALC then computes the following plan (Plan 1) of length 11 for this problem:

```
0:
1: move(r1,right,steps=1)
   move(r2,right,steps=1)
2: move(r1,right,steps=1)
   move(r2,right,steps=1)
3: move(r1,up,steps=1)
   move(r1,right,steps=1)
   move(r2,right,steps=4)
.
.
.
10: move(r1,right,steps=1)
    move(r1,down,steps=1)
    move(r2,right,steps=1)
    move(r2,down,steps=1)
```

CCALC computes Plan 1 in the style of satisfiability planning (Kautz and Selman 1992):

1. transforms the domain description into a propositional theory Γ_D,
2. transforms the planning problem P into a propositional theory Γ_P,
3. computes a satisfying interpretation for $\Gamma_D \cup \Gamma_P$, and
4. extracts a plan from the satisfying interpretation.

A detailed description of each step above can be found in (Giunchiglia and Lifschitz 2004).

Finding a Collision-Free Task Plan

The constraints included in the action domain description above ensure at each step that the length of a payload does not change more than a specified tolerance, and the robot

Algorithm 1 PLAN

Input: An action domain description $\mathcal{D}$, a planning problem $\mathcal{P}$
Output: A collision-free plan P of length at most n, if exists

```
plan := false;    // no collision-free plan
while ¬plan do
  plan, P, H ← Compute a plan P of length at most n, within a history H, using CCALC with D and P, if there exists such a plan;
  if plan then
    collision := false;    // no trajectory collision
    i := 0;
    while ¬collision AND i ≤ |P| do
      Δ ← Extract the relevant parameters from the history H to uniquely identify the positions of the robot end-effectors at Steps i and i + 1;
      // Extract the location L of the payload and the action A executed at Step i, if a collision is detected
      collision, L, A ← trajectoryCollision(Δ);
      i++;
    end while
    if ¬collision then
      return P
    else
      P ← Modify the planning problem P to compute a plan that does not execute A at a state where the payload is located at L;
      plan := false;
    end if
  end if
end while
```

end-effectors and the payload do not collide with an obstacle. However, during the plan execution, between any two steps of the plan, the length of a payload can change out of the specified range, and there may be collisions. To ensure collision-free trajectories for the robot end-effectors and the payload, a collision detection algorithm is required. Such an algorithm is described in Algorithm 1. The idea is to compute a plan with the given action description $\mathcal{D}$ using CCALC, and then to check whether such a plan could lead to a trajectory collision. If such a collision is detected between Steps i and $i+1$, then we extract the location L of the payload and the action A executed at Step i and ask CCALC for a different plan that does not execute A at a state where the payload is located at L.

Consider, for instance, the planning problem described in the previous section. According to Plan 1, at Step 2, each robot moves up by 1 unit and moves right by 1 unit at the same time. However, while executing this plan, between time units 3 and 4, the payload collides with the obstacle as illustrated in Fig. 2. Therefore, from the history CCALC computed, we extract the position L of the payload at Step 3:

```
xpay(pl1)=2 xpay(pl2)=2
ypay(pl1)=0 ypay(pl2)=5
```

and the actions A executed at Step 3:

```
move(r1,up,steps=1) move(r1,right,steps=1)
move(r2,right,steps=4)
```

After that, we ask CCALC to find a different plan that does not execute the actions A at a state where the payload is located at L, by modifying the query above as follows:

```
:- query
maxstep :: 10..infinity;
0: -canCarry, xpos(r1)=0, ypos(r1)=0,
   xpos(r2)=0, ypos(r2)=5,
   xpay(pl1)=0, ypay(pl1)=0,
   xpay(pl2)=0, ypay(pl2)=5;
maxstep: xpay(pl1)=10, ypay(pl1)=0,
   xpay(pl2)=10, ypay(pl2)=5;
T<maxstep ->> (
  ((T: xpay(pl1)=2) && (T: ypay(pl1)=0) &&
   (T: xpay(pl2)=2) && (T: ypay(pl2)=5))
  ->>
   -((T: move(r1,up)) &&
     (T: steps(r1,up)=1) &&
     (T: move(r1,right)) &&
     (T: steps(r1,right)=1) &&
     (T: move(r2,right)) &&
     (T: steps(r2,up)=4) )).
```

By this way, at the 9'th iteration of Algorithm 1, CCALC

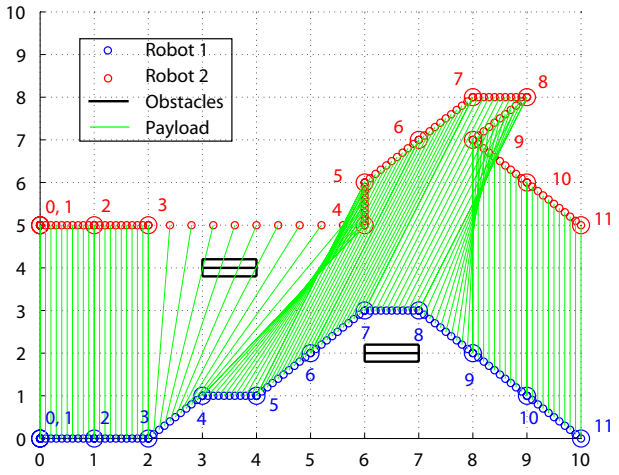

Figure 2: This figure illustrates the execution of Plan 1 on the robot system. Colors blue, red, green and black are associated with the Robots 1 and 2, the payload, and the obstacles, respectively. Circles and their labels indicate the positions of the robot end-effectors. The green lines denote the position of the payload: the ones connecting larger circles are obtained from the history calculated by CCALC, whereas the others are constructed from the motor encoder data. For instance, according to the computed history, at Step 3, the end-effectors of Robots 1 and 2 are located at (2,0) and (2,5) respectively, holding the end points of the payload. Observe that, although at time steps 3 and 4 the payload does not collide with the obstacles, between time steps 3 and 4 it does collide with the obstacles. Also the length of the payload changes more than the allowable tolerance.

computes the following collision-free plan (Plan 2) of length 11 for this problem:

```
0:
1: move(r1,up,steps=1)
   move(r2,up,steps=1)
   move(r2,right,steps=1)
2: move(r1,up,steps=1)
   move(r1,right,steps=1)
   move(r2,up,steps=1)
   move(r2,right,steps=1)
3: move(r1,up,steps=1)
   move(r1,right,steps=1)
   move(r2,up,steps=1)
   move(r2,right,steps=1)
.
.
.
10: move(r1,right,steps=1)
    move(r2,down,steps=1)
    move(r2,right,steps=1)
```

Recall that CCALC computes a plan, such as Plan 2, in four steps, as described above. In this example, CCALC transforms the domain description and the planning problem into a propositional theory (Steps 1 and 2) in 48.5 CPU seconds, and then computes Plan 2 (Steps 3 and 4) at the 9'th iteration in 6.8 CPU seconds.[6] Here the propositional theory $\Gamma_D \cup \Gamma_P$ consists of 12491 variables and 271782 clauses. It is important to emphasize here that the grounding of the domain description (Step 1) is performed by CCALC only once, at the very first iteration of Algorithm 1.

Finding a Collision-Free Trajectory Plan

Once CCALC computes a plan for a given problem, it logs the complete history (including the state information). From such a plan, the positions of the robot end-effectors at each time step can be extracted. The simplest approach for executing the plan would be to convert these state values into motor angles and use these values as set-point references for the motors. However, set-point tracking does not guarantee a linear motion of the end-effector, and may cause collisions with the obstacles. To obtain more straight trajectories, a simplified trajectory tracking controller is implemented by introducing intermediate steps to the plan using linear interpolation. Then, these intermediate points are mapped to robot joint variables.

Fig. 3 depicts a schematic representation of two planar robots carrying a payload. For each robot i, its end-effector is located at a grid point (x_i,y_i) and its corresponding joint variables are denoted as (s_i,θ_i). The forward kinematics of each robot maps its joint variables to its end-effector coordinates and reads as

$$x_i = s_i + l_i \cos(\theta_i) \quad (1)$$

$$y_i = l_i \sin(\theta_i) \quad (2)$$

[6]The CPU times are for a PC with 1.66 GHz Celeron processor, 2x512 MB RAM, and Ubuntu 9.04 Linux. We have used CCALC (Version 2.0) with SWI-Prolog (Version 5.6.64) and MiniSAT (Version 2.0).

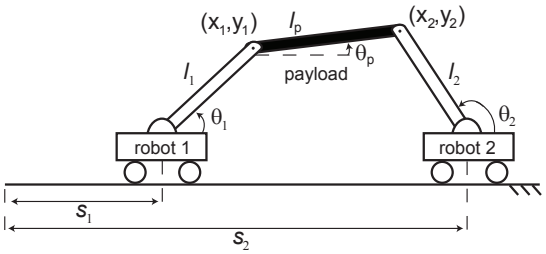

Figure 3: Schematic representation of two robots carrying a payload.

Algorithm 2 NXC Program

Input: Trajectories (a list of reference angles)
Output: Log file
Check for the Bluetooth communication
Go to the initial configuration
Wait for the start signal
while There is a trajectory to follow **do**
  Read the reference angles
  while Not at the reference angles **do**
    Read the motor angles
    Calculate the error in motor position
    Rotate the motor to compensate for the error
    Record the motor angles
  end while
end while

while the inverse kinematics maps the end-effector coordinates to the joint variables and is given as

$$s_i = x_i \pm \sqrt{l_i^2 - y_i^2} \quad (3)$$

$$\theta_i = atan2\left(\pm\sqrt{l_i^2 - y_i^2}, y_i\right) \quad (4)$$

where l_i represents the length of each robot arm. One can observe that two feasible solutions exist for the inverse kinematics of each robot and the $\pm$ signs in equations (3) and (4) are coupled.

Executing a Plan on LEGO Robots

After the joint space trajectories are calculated, they are passed to the robots, by means of messages via Bluetooth communication, using the NeXTTool program. Based on these joint space trajectories, the computed plan is executed by the robots via an NXC program. Algorithm 2 presents the structure of the NXC program used for the low level control of the robots.

To locate a robot at a reference configuration within an acceptable error margin, it is essential that the actual configuration of the robot is checked with respect to the reference configuration. Hence, a feedback controller is necessitated. Due to its ease of implementation, a proportional feedback controller (P-controller) is employed to ensure a robust tracking of the robots in the joint space. The P-controller continually compares the reference and actual joint variables

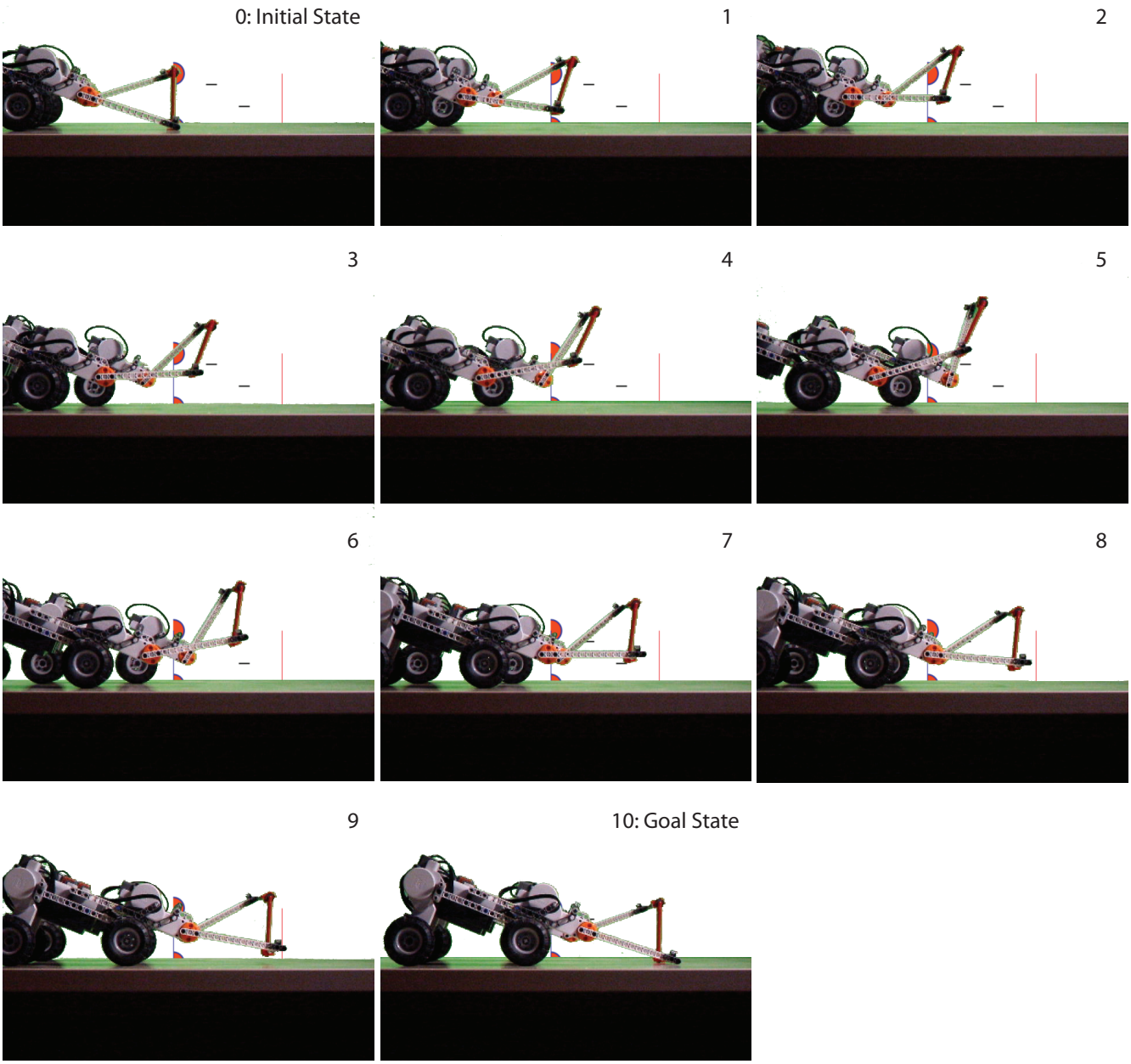

Figure 4: Snapshot taken at each step of the plan.

and compensates for the error term by commanding a counteracting motion that is proportional to the magnitude of the error signal. The P-controller gain is tuned empirically to achieve acceptably low overshoot and steady state error of the motor response.

For instance, consider the planning problem described in the previous section. After CCALC computes a collision-free plan (Plan 2) for the problem as described in the previous sections, the intermediate points are interpolated and mapped to the robot joint space as explained above. Then, the LEGO robots trace these trajectories as in Fig.s 4 and 5. Fig. 4 presents snapshots taken at each step of the plan, while Fig. 5 depicts the trajectories of the robot end-effectors and the payload.

Discussion

We have demonstrated with some planning problems in a sample action domain, with concurrency and cost constraints, how the logic-based formalism $\mathcal{C}+$ can be used to endow two LEGO MINDSTORMS NXT robots with high-level reasoning in the style of cognitive robotics, by transforming a discrete task plan into a collision-free continuous

trajectory.

The action domain considered involves concurrent execution of actions, direct effects of actions on additive fluents, ramifications of actions, qualification constraints, and the frame problem. These challenges can be handled easily within $\mathcal{C}+$ and using the reasoner CCALC, unlike the other logic-based reasoning systems (mentioned in the introduction) and the state-of-the-art planners.

LEGO MINDSTORMS NXT robots are available at a relatively low price and widely available all over the world compared to more sophisticated robots. They can be built easily in various forms; the availability of the software systems and the programming languages (with good documentation) allow us to implement programs to control these robots.

Although the language $\mathcal{C}+$ and using the reasoner CCALC provided us a formal framework expressive in many ways, and LEGO MINDSTORMS NXT robots provided us an inexpensive platform to test our ideas, we encountered many challenges. For instance, that CCALC can handle integers only, caused some difficulties in calculating the exact positions of the robots. To deal with this problem, we assumed that the length of the payload might increase/decrease within a specified tolerance.

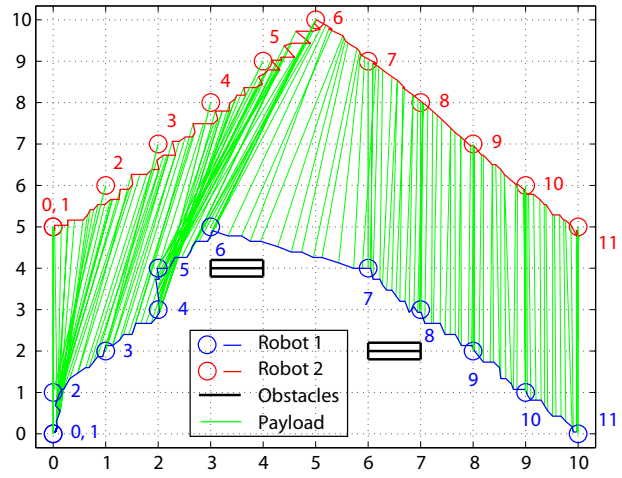

Figure 5: This figure illustrates the execution of the collision free plan (Plan 2) on the robot system. Colors blue, red, green and black are associated with the Robots 1 and 2, the payload, and the obstacles, respectively. Circles and their labels indicate the positions of the robot end-effectors. The green lines denote the position of the payload: the ones connecting larger circles are obtained from the history calculated by CCALC, whereas the others are constructed from the motor encoder data. For instance, according to the computed history, initially, the end-effectors of robots are located at the grid point (0,0) and (0,5) respectively, holding the end points of the payload; at Step 11, the end-effectors of robots are located at (10,0) and (10,5) respectively, holding the end points of the payload. Blue and red lines represent the end-effector trajectories of each robot, while thinner green lines denote the payload configuration as constructed from the motor encoder data. The black lines represent the obstacles.

We also faced control challenges: Lack of floating point operations in NXC; low encoder resolution, high friction and backlash of the LEGO motors; and the flexible robot structure due to plastic parts. To address these challenges we have to upgrade the hardware/software of LEGO MINDSTORMS NXT robots.

The modification of the overall architecture to include execution monitoring is a part of the ongoing work.

Acknowledgments

Thanks to anonymous reviewers for helpful comments. This work has been partially supported by Sabancı University IRP Grant.

References

Caldiran, O.; Haspalamutgil, K.; Ok, A.; Palaz, C.; Erdem, E.; and Patoglu, V. 2009. Bridging the gap between high-level reasoning and low-level control. In *Proc. of LPNMR*.

Doherty, P.; Gustafsson, J.; Karlsson, L.; and Kvarnström, J. 1998. Tal: Temporal action logics language specification and tutorial. *ETAI* 2:273–306.

Doherty, P.; Granlund, G.; Kuchcinski, K.; Sandewall, E.; Nordberg, K.; Skarman, E.; and Wiklund, J. 2000. The WITAS unmanned aerial vehicle project. In *Proc. of ECAI*, 747–755.

Erdem, E., and Gabaldon, A. 2005. Cumulative effects of concurrent actions on numeric-valued fluents. In *Proc. AAAI*, 627–632.

Ferrein, A.; Fritz, C.; and Lakemeyer, G. 2005. Using GOLOG for deliberation and team coordination in robotic soccer. *Künstliche Intelligenz* 1.

Fichtner, M.; Großmann, A.; and Thielscher, M. 2003. Intelligent execution monitoring in dynamic environments. In *Proc. of Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modeling, planning, learning, and communicating*.

Finger, J. 1986. *Exploiting Constraints in Design Synthesis*. Ph.D. Dissertation, Stanford University. PhD thesis.

Giunchiglia, E., and Lifschitz, J. L. V. 2004. Nonmonotonic causal theories. *AIJ* 153:2004.

Hähnel, D.; Burgard, W.; and Lakemeyer, G. 1998. GOLEX - bridging the gap between logic (GOLOG) and a real robot. In *Proc. of KI*, 165–176.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. ECAI-92*, 359–363.

Kowalski, R., and Sergot, M. 1986. A logic-based calculus of events. *New Gen. Comput.* 4(1):67–95.

Lee, J., and Lifschitz, V. 2003. Describing additive fluents in action language $\mathcal{C}+$. In *Proc. IJCAI-03*.

Levesque, H., and Lakemeyer, G. 2007. Cognitive robotics. In *Handbook of Knowledge Representation*. Elsevier.

Levesque, H. J., and Pagnucco, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *Proc. of CogRob*, 104–109.

Levesque, H. J.; Reiter, R.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *JLP* 31.

Levesque, H. J.; Pirri, F.; and Reiter, R. 1998. Foundations for the situation calculus. *ETAI* 2:159–178.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence*, volume 4. Edinburgh: Edinburgh University Press. 463–502.

McCarthy, J. 1963. Situations, actions, and causal laws. Technical report, Stanford University.

McCarthy, J. 1980. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence* 13:27–39,171–172.

Miller, R., and Shanahan, M. 1999. The event calculus in classical logic - alternative axiomatisations. *ETAI* 3(A):77–105.

Sandewall, E. 1994. *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems*. Oxford University Press.

Sandewall, E. 1998. Cognitive robotics logic and its metatheory: Features and fluents revisited. *ETAI* 2:307–329.

Shanahan, M., and Witkowski, M. 2000. High-level robot control through logic. In *ATAL*, 104–121.

Thielscher, M. 1998. Introduction to the fluent calculus. *ETAI* 2:179–192.

Thielscher, M. 2005. FLUX: A logic programming method for reasoning agents. *TPLP* 5(4-5):533–565.