Bridging the Gap between High-Level Reasoning and Low-Level Control

Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu

Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul, Turkey

Abstract. We present a formal framework where a nonmonotonic formalism (the action description language $\mathcal{C}+$) is used to provide robots with high-level reasoning, such as planning, in the style of cognitive robotics. In particular, we introduce a novel method that bridges the high-level discrete action planning and the low-level continuous behavior by trajectory planning. We show the applicability of this framework on two LEGO MINDSTORMS NXT robots, in an action domain that involves concurrent execution of actions that cannot be serialized.

1 Introduction

As robotics technology is broadening its applications from factory to more general-purpose applications in public use, demands from robots shift from speed and precision towards safety and cognition. New levels of robustness, physical dexterity, and cognitive capability are necessitated from robots that can perform in dynamic environments involving humans. While traditional robotics design and construct extremely rigid robots with high position control gains, cognitive robotics [1] is concerned with providing robots with higher level cognitive functions that involve reasoning about goals, perception, actions, the mental states of other agents, collaborative task execution, etc., so that they can give high-level decisions to act intelligently in a dynamic world. This paper is an attempt to close the gap between traditional robotics and cognitive robotics, to meet the demands of various applications from robots.

There have been various studies to close the gap between traditional robotics and cognitive robotics, by implementing high-level robot control systems based on different families of formalisms for reasoning about actions and change. For instance, [2] describes a system, LEGOLOG[1], that controls a LEGO MINDSTORMS RIS robot using the high-level control language GOLOG [3] based on the situation calculus [4,5]. [6] presents an execution monitoring system for GOLOG and the RHINO control software which operates on RWI B21 and B14 mobile robots. [7] studies coordination of soccer playing robots, using an extension of GOLOG. In the WITAS Unmanned Aerial Vehicle Project[2] temporal action logic [8], features and fluents [9], and cognitive robotics logic [10] are used for representing the actions and the events, as a part of a helicopter control system [11]. [12] describes how event calculus [13,14] can be used to provide

[1] `http://www.cs.toronto.edu/cogrobo/Legolog`
[2] `http://www.ida.liu.se/ext/witas`

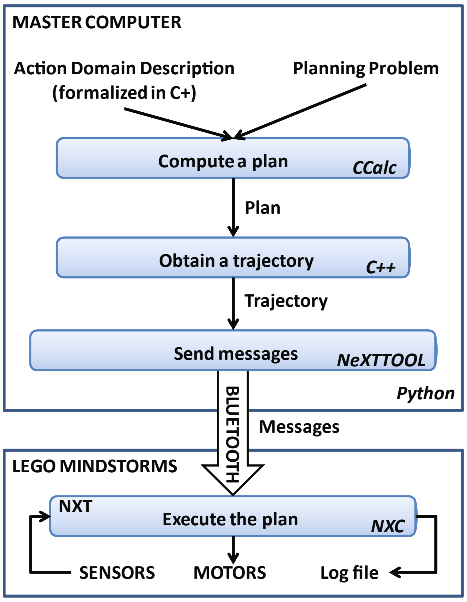

Fig. 1. The overall system architecture

high-level control for a Khepera robot. The agent programming language FLUX [15], based on the fluent calculus [16], has also been used to control the execution of some robots.[3] For instance, [17] presents how FLUX can be used for monitoring the execution of a plan, on a Pioneer 2 mobile robot.

We continue this line of research to provide traditional robotics with high-level reasoning in the style of cognitive robotics, with a different formalism (i.e., the action description language $\mathcal{C}+$ [18]), a different reasoner (i.e., CCALC[4]), a different robot (i.e., LEGO MINDSTORMS NXT), and most importantly with a different method, which bridges the high-level discrete action planning and the low-level continuous behavior with "trajectory planning".

Also we consider high-level reasoning in a different sort of action domain that involves concurrent execution of actions that cannot be serialized. 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. The idea is for the robots to automatically generate a plan, and then execute it collaboratively (Fig. 1). In this domain, the robots can follow complex paths (not necessarily a straight path marked a priori, as in LEGOLOG) avoiding obstacles; this is why our system has trajectory planning between the high-level discrete action planning and the low-level continuous behavior. We can describe this action domain (in particular, the frame problem, the ramification problem, the qualification problem, nonserializable concurrency) in $\mathcal{C}+$ in a straightforward way.

[3] `http://www.fluxagent.org/projects.htm`
[4] `http://www.cs.utexas.edu/users/tag/cc`

Our main contributions can be summarized in two parts:

- Action description languages [19] are well-studied for various sorts of high-level reasoning about actions and change. On the other hand, unlike the other formalisms mentioned above, it has not been shown on a real robot how high-level reasoning performed within an action description language can be useful for traditional robotics. In that sense, our work is the first to demonstrate the use of action description languages for high-level reasoning and control of robots, in the style of cognitive robotics.
- LEGO MINDSTORMS NXT is available at a relatively low price and is widely available all over the world compared to more sophisticated robots. It allows one to build various kinds of robots, and write programs to control them. Also, the high-level reasoning component based on the action description language $\mathcal{C}+$ can be replaced by one based on a different formalism. These features of the overall system enable the reproduction and improvement of our work for educational and research purposes by other researchers who study action description languages, other formalisms for reasoning about actions and change, cognitive robotics, and also traditional robotics.

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.

2 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 $\mathcal{C}+$ [18]. 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 plan to reach a goal state, and displays the complete history (including the state information). From such a history, we extract the 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.

3 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. The goal is to place the payload at a specified goal position on the platform.

3.1 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.

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.

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).
```

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 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.

3.2 Collision Detection

The constraints above ensure at each step that the length of a payload does not change more than a specified tolerance, and the robot 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 a collision detection algorithm can be implemented in C++ as a function, which takes as inputs the end-effector coordinates of a robot (or the end point coordinates of the payload) at the current state and the next state, and returns "true" if the path is free of collisions. Let's call this function `trajectoryCollision`. After that, we can prevent collision by adding to the description in Section 3.1 the causal laws

```
caused false
  if xpay(pl1)=X1 & ypay(pl1)=Y1 & xpay(pl2)=X2 & ypay(pl2)=Y2
  after
    xpay(pl1)=X3 & ypay(pl1)=Y3 & xpay(pl2)=X4 & ypay(pl2)=Y4
  where trajectoryCollision(X1,Y1,X3,Y3,X2,Y2,X4,Y4) &
    between(X3-maxN,X3+maxN,X1) & between(Y3-maxN,Y3+maxN,Y1) &
    between(X4-maxN,X4+maxN,X2) & between(Y4-maxN,Y4+maxN,Y2).
```

However, there are too many of such causal laws (due to the 8 schematic variables). (Grounding the schematic law above takes more than an hour if we reduce the grid size and restrict the collision detection check to a small set of possible positions of the robot end-effectors.) We can modify `trajectoryCollision` so that it takes 6 variables instead (the positions of one of the end points and the orientation of the payload at the current state and the next state) to uniquely determine the current and the next positions of the robot end-effectors and the payload; add new definitions for the orientations; and modify the schematic causal law above. However, these modifications do not reduce the grounding time sufficiently.

Therefore, instead of adding to the action domain description $\mathcal{D}$ in Section 3.1 causal laws with too many variables, we apply Algorithm 1. The idea is to compute a plan with $\mathcal{D}$ using CCALC, and then 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. It is important to note that CCALC grounds the action domain only once at the very first iteration of the algorithm; after that, no grounding is done to compute a collision-free plan.

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 is computed so far
  while $\neg plan$ do
    $plan, P, H \leftarrow$ Compute a plan $P$ of length at most $n$, within a history $H$, using CCALC
    with $\mathcal{D}$ and $\mathcal{P}$, if there exists such a plan;
    if $plan$ then
      $collision := false$;   // no trajectory collision is detected so far
      $i := 0$;
      while $\neg collision$ AND $i \leq |P|$ do
        $\Delta \leftarrow$ 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
        $collision, L, A \leftarrow trajectoryCollision(\Delta)$;
        $i++$;
      end while
      if $\neg collision$ then
        return $P$
      else
        $\mathcal{P} \leftarrow$ Modify the planning problem $\mathcal{P}$ 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
```

4 Finding a Collision-Free Plan

Suppose that initially the robots `r1` and `r2` are at `(1,1)` and `(2,1)` respectively, and the end points of the payload are at `(4,1)` and `(9,1)`. The goal is to move the payload to a location so that its end points are at `(4,9)` and `(9,9)`. 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)=1, ypos(r1)=1, xpos(r2)=1, ypos(r2)=1,
   xpay(pl1)=4, ypay(pl1)=1, xpay(pl2)=9, ypay(pl2)=1;
maxstep: xpay(pl1)=9, ypay(pl1)=9, xpay(pl2)=4, ypay(pl2)=9.
```

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

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

```
   move(r2,right,steps=2)
4:
5: move(r2,up,steps=3) move(r2,left,steps=1)
6: move(r1,up,steps=2) move(r1,right,steps=2)
   move(r2,up,steps=3) move(r2,right,steps=1)
7: move(r1,up,steps=2) move(r1,right,steps=3)
   move(r2,up,steps=2) move(r2,left,steps=3)
8: move(r1,up,steps=4) move(r2,left,steps=2)
```

However, while executing this plan, between time units 7 and 8, 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 7:

```
xpay(pl1)=6 xpay(pl2)=9 ypay(pl1)=3 ypay(pl2)=7
```

and the actions A executed at Step 7:

```
move(r1,up,steps=2) move(r1,right,steps=3)
move(r2,up,steps=2) move(r2,left,steps=3)
```

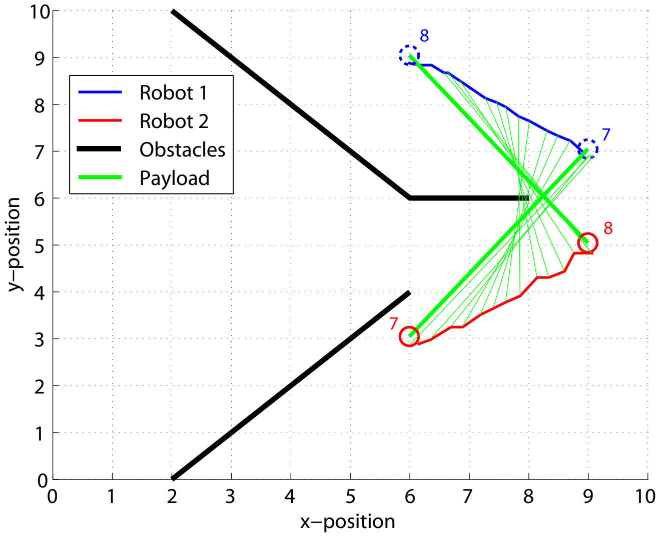

Fig. 2. This figure illustrates the execution of Plan 1 on the robot system, at time step 7. Colors blue, red, green and black are associated with the Robots 1 and 2, the payload, and the obstacles, respectively. Circles and their labels indicate the positions of the robot end-effectors, while the thick green lines denote the position of the payload at each step according to the history calculated by CCALC. For instance, according to the computed history, at Step 7, the end-effectors of Robots 1 and 2 are located at (6,3) and (9,7) respectively, holding the end points of the payload. The thinner green lines denote the payload configuration constructed from the motor encoder data. Observe that, although at time steps 7 and 8 the payload does not collide with the obstacles, between time steps 7 and 8 it does collide with the obstacles. Also the length of the payload changes more than the allowable tolerance.

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
label::5;
maxstep :: 9..infinity;
0: -canCarry, xpos(r1)=1, ypos(r1)=1, xpos(r2)=1, ypos(r2)=1,
   xpay(pl1)=4, ypay(pl1)=1, xpay(pl2)=9, ypay(pl2)=1;
maxstep: xpay(pl1)=9, ypay(pl1)=9, xpay(pl2)=4, ypay(pl2)=9;
T<maxstep ->> (
   ((T: xpay(pl1)=6) && (T: ypay(pl1)=3) &&
    (T: xpay(pl2)=9) && (T: ypay(pl2)=7)) ->>
   -((T: move(r1,up)) && (T: steps(r1,up)=2) &&
     (T: move(r1,right)) && (T: steps(r1,right)=3) &&
     (T: move(r2,up)) && (T: steps(r2,up)=2) &&
     (T: move(r2,left)) && (T:steps(r2,left)=3) )).
```

Then CCALC computes the following plan (Plan 2)

```
0: move(r1,up,steps=2) move(r1,right,steps=3)
   move(r2,up,steps=2) move(r2,right,steps=3)
1: move(r1,up,steps=2) move(r1,right,steps=3)
   move(r2,up,steps=2) move(r2,right,steps=3)
2: move(r1,down,steps=1) move(r1,right,steps=2)
   move(r2,down,steps=3) move(r2,left,steps=1)
3: move(r1,down,steps=3) move(r2,down,steps=1) move(r2,left,steps=2)
4:
5: move(r1,up,steps=4) move(r2,up,steps=1) move(r2,right,steps=1)
6: move(r1,up,steps=4) move(r2,up,steps=2) move(r2,right,steps=3)
7: move(r1,left,steps=3) move(r2,up,steps=1) move(r2,right,steps=1)
8: move(r1,left,steps=2) move(r2,up,steps=4)
```

According to this plan, for instance, at Step 7, Robot 1 moves left by 3 units, and Robot 2 moves up by 1 unit and right by 1 unit.

5 Executing a Plan on LEGO Robots

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.

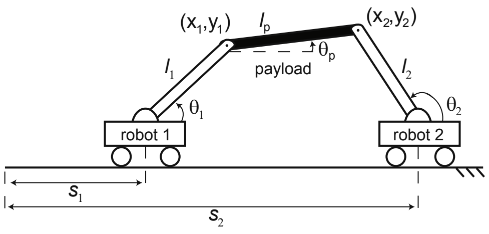

Fig. 3. Schematic representation of two robots carrying a payload

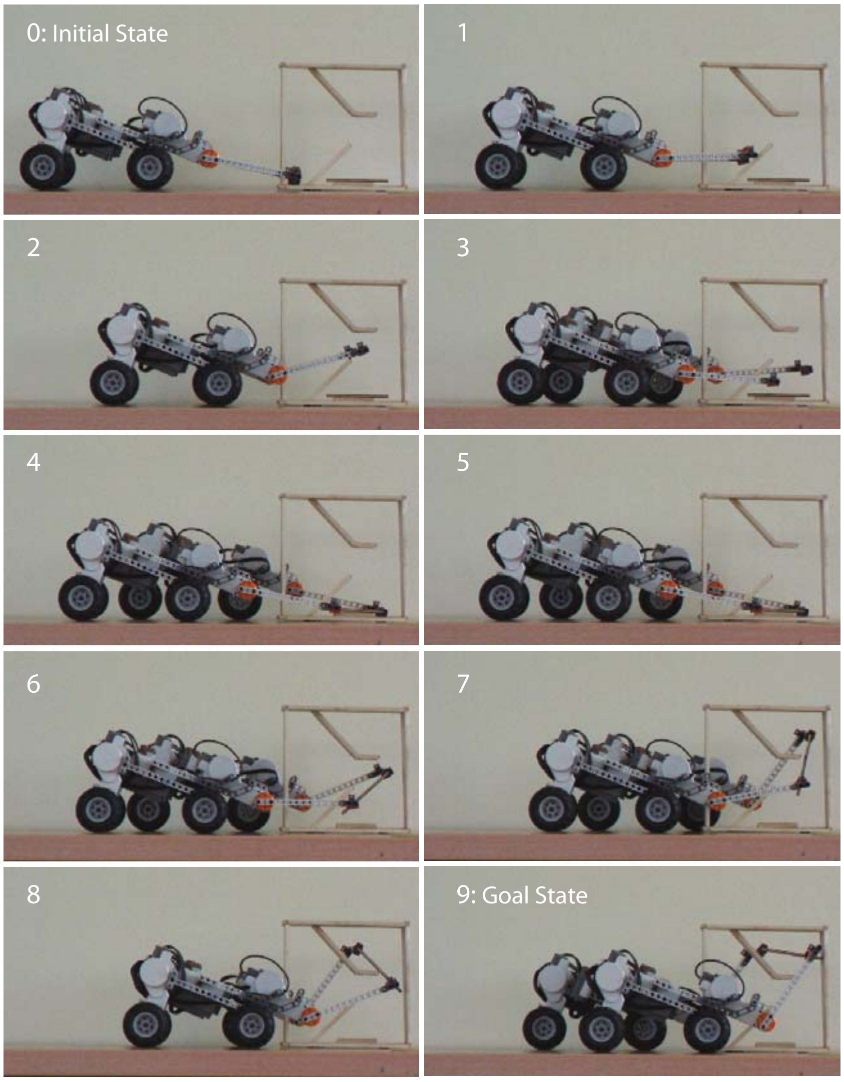

Fig. 4. Snapshot taken at each step of the plan

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

$$x_i = s_i + l_i \cos(\theta_i) \tag{1}$$
$$y_i = l_i \sin(\theta_i) \tag{2}$$

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

$$s_i = x_i \pm \sqrt{l_i^2 - y_i^2} \tag{3}$$
$$\theta_i = atan2\left(\pm\sqrt{l_i^2 - y_i^2}, y_i\right) \tag{4}$$

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.

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 and compensates for the error term by

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
```

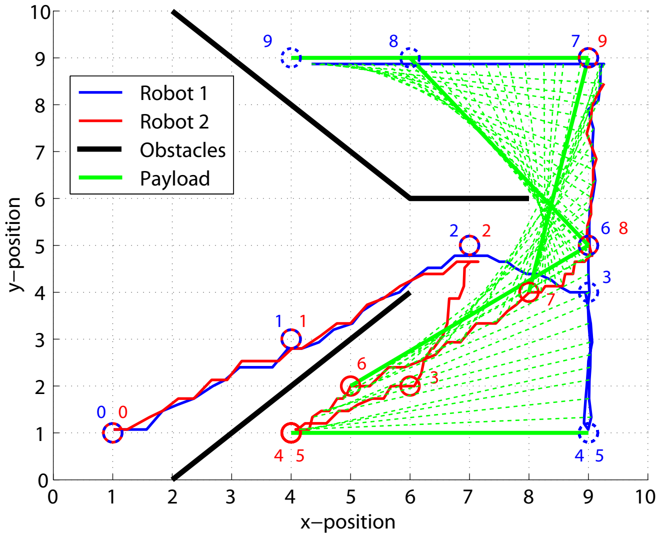

Fig. 5. This figure illustrates the execution of Plan 2 on the robot system. Colors blue, red, green and black are associated with the Robots 1 and 2, the payload, and the obstacles, respectively. Circles and their labels indicate the positions of the robot end-effectors, while the thick green lines denote the position of the payload at each step according to the history calculated by CCalc. For instance, according to the computed history, initially, the end-effectors of robots are located at the grid point (1,1), and the payload lies between the points (4,1) and (9,1); at Step 6, the end-effectors of Robots 1 and 2 are located at (9,5) and (5,2) 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.

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.

6 Discussion

We have demonstrated with some planning problems in a sample action domain, 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.

In these experiments, 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. 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 monitoring of the plan execution is a part of the ongoing work.

References

1. Levesque, H., Lakemeyer, G.: Cognitive robotics. In: Handbook of Knowledge Representation. Elsevier, Amsterdam (2007)
2. Levesque, H.J., Pagnucco, M.: Legolog: Inexpensive experiments in cognitive robotics. In: Proc. of CogRob, pp. 104–109 (2000)
3. Levesque, H.J., Reiter, R., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. JLP 31 (1997)
4. McCarthy, J.: Situations, actions, and causal laws. Technical report, Stanford University (1963)
5. Levesque, H.J., Pirri, F., Reiter, R.: Foundations for the situation calculus. ETAI 2, 159–178 (1998)
6. Hähnel, D., Burgard, W., Lakemeyer, G.: GOLEX - bridging the gap between logic (GOLOG) and a real robot. In: Herzog, O. (ed.) KI 1998. LNCS, vol. 1504, pp. 165–176. Springer, Heidelberg (1998)
7. Ferrein, A., Fritz, C., Lakemeyer, G.: Using GOLOG for deliberation and team coordination in robotic soccer. Künstliche Intelligenz 1 (2005)
8. Doherty, P., Gustafsson, J., Karlsson, L., Kvarnström, J.: Tal: Temporal action logics language specification and tutorial. ETAI 2, 273–306 (1998)
9. Sandewall, E.: Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems. Oxford University Press, Oxford (1994)
10. Sandewall, E.: Cognitive robotics logic and its metatheory: Features and fluents revisited. ETAI 2, 307–329 (1998)
11. Doherty, P., Granlund, G., Kuchcinski, K., Sandewall, E., Nordberg, K., Skarman, E., Wiklund, J.: The WITAS unmanned aerial vehicle project. In: Proc. of ECAI, pp. 747–755 (2000)
12. Shanahan, M., Witkowski, M.: High-level robot control through logic. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, pp. 104–121. Springer, Heidelberg (2001)
13. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gen. Comput. 4(1), 67–95 (1986)
14. Miller, R., Shanahan, M.: The event calculus in classical logic - alternative axiomatisations. ETAI 3(A), 77–105 (1999)
15. Thielscher, M.: FLUX: A logic programming method for reasoning agents. TPLP 5(4-5), 533–565 (2005)
16. Thielscher, M.: Introduction to the fluent calculus. ETAI 2, 179–192 (1998)
17. Fichtner, M., Großmann, A., Thielscher, M.: 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, Acapulco, Mexico (2003)
18. Giunchiglia, E., Lifschitz, J.L.V.: Nonmonotonic causal theories. AIJ 153 (2004)
19. Gelfond, M., Lifschitz, V.: Action languages. ETAI 2, 193–210 (1998)