ev3dev-prolog – Prolog API for LEGO EV3

Sibylle Schwarz[1] Mario Wenzel[2]

Abstract: We present ev3dev-prolog – an extendable Prolog API to control LEGO EV3 robots – and demonstrate our approach by several examples from introductory robotics courses like obstacle avoidance and Braitenberg vehicles as well as a more complex example. We show how to interleave our API with planning and replanning in Prolog to move a robot through an unknown environment.

The presented API is divided into two abstraction layers. Low level predicates control individual sensors and actors, higher level predicates control user-defined robots consisting of several sensors and actors. The connection between the parts of the robot and an SWI-Prolog interpreter running on the robot is established via ev3dev.

Keywords: logic programming, SWI-Prolog, robotics, LEGO EV3, ev3dev

1 Motivation

Intelligent control of autonomous robots has been an important goal of AI research from its very beginning. Applications in mobile robotics are frequently used to motivate research and lectures on AI, planning and logic programming [Po95], [RN95]. However, in many AI courses this motivation remains theoretical.

On the other hand, robotics experiments for beginners are evidently successful in growing young people's interest in STEM topics (Science, Technology, Engineering, Mathematics) like computer science, construction, physics and mechanics. LEGO MINDSTORMS EV3 is a flexible robotics platform for construction and programming of various robots with standard LEGO pieces. These robots are frequently used in introductory robotics courses for children and young students. Since LEGO MINDSTORMS robots allow advanced constructions with complex behaviour, they are also used in university courses and in research.

LEGO provides a visual programming environment to control EV3 robots particularly suited for beginners. For several reasons, we even use the visual programming interface in robotics projects and competitions for our first year students. Instructions for many basic experiments in robotics like obstacle avoidance, line following, self-balancing, are available online and in the literature.

[1] HTWK Leipzig, Institut für Informatik, sibylle.schwarz@htwk-leipzig.de
[2] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, mario.wenzel@informatik.uni-halle.de

doi:10.18420/inf2019_ws41

Due to the reactive nature of commands to control sensors and actors, they are usually implemented as side effects in low level imperative languages. Over the years, several platform-independent and open source APIs for imperative languages evolved, like Java [Lu16] or LEGOs special C-like language NXC.

Logic programming languages like Prolog allow concise and clear descriptions of desired properties of solutions without explicitly describing how to obtain a solution. Logic programming is particularly suitable for AI tasks containing nondeterministic search and rule processing.

To apply these advantages to robot control, bindings between logic programming languages and lower level imperative sensor and actor commands are necessary. Unfortunately, bindings between robot control and logic programming are rare. There have been some successful approaches to connect Prolog and AI to predecessors of the LEGO EV3 system. In this paper, we further develop this idea to a connection between SWI-Prolog and the EV3 hardware. We have not yet tested our system with other Prolog systems.

Legolog [LP00] is a Prolog-based system for LEGO RCX robots. This system involves the Golog planner [Le97] and interleaves generation and execution of plans. We adopt this approach of interleaved planning and moving in unknown environments in our experiment in Section 6 and present a solution for EV3 robots using our Prolog API ev3dev-prolog. In [HH02], Hanus and Höppner provide a framework to program LEGO RCX robots in the functional-logic language Curry. In [Na08], Nalepa presents a Prolog API for LEGO NXT robots that contains predicates for basic sensor and motor actions. The predicates in the lower layer of ev3dev-prolog resemble these predicates. In some cases, we use fewer parameters since connections between motors, sensors and EV3 ports can be recognized automatically. Moreover, ev3dev-prolog contains a second abstraction layer with predicates to define robot configurations and control robots consisting of several sensors and actors.

We use the alternative operating system ev3dev for EV3 robots. It wraps all communication details between actors, sensors, and the CPU. With ev3dev-prolog, we provide a method to connect Prolog-based AI with EV3 robots, both at a level accessible to non-experts in robotics and Prolog. We demonstrate the application of ev3dev-prolog in examples from our introductory robotics courses for students.

Section 2 contains a short overview of the EV3 hardware and the ev3dev operating system.

In Section 3, we explain the implementation of Prolog predicates to control sensors and actors of the EV3 system. We show how to use this low level part of ev3dev-prolog to control Braitenberg vehicles, simple robots with direct sensor actor interaction.

In Section 4, we show how to control robots consisting of several parts, i.e. motors, wheels, and sensors in a fixed configuration by higher level predicates – like “move 200 mm” or “turn 30 deg”. The translation of these commands to low level commands depends on several parameters like wheel diameter and distance of the robot. We present predicates to define configurations of robots and demonstrate how to move a robot and avoid dynamic obstacles using a combination of higher and lower level predicates in ev3dev-prolog.

In Section 5, we summarize the high level predicates in ev3dev-prolog as an interface to Prolog programmers that are not interested in the details of the driver framework.

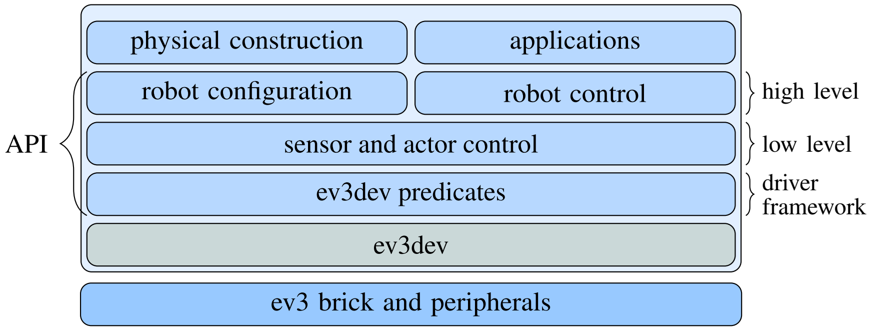
Section 6 contains a complex example that uses ev3dev-prolog in a combination of robotics and AI. We present a Prolog program to control a robot with a more complex behaviour that involves path planning in partially unknown environments.

2 ev3dev-prolog – A Prolog Binding for LEGO EV3

EV3 [Le] is the most recent LEGO MINDSTORMS robotics platform. It consists of a programmable brick, motors and sensors that can be combined by standard LEGO pieces to a wide variety of robots. Unlike its predecessors, the EV3 brick includes a powerful ARM9 CPU processor running Debian Linux. Hence advanced programs like Prolog interpreters can be installed and run directly on the brick.

For LEGO EV3 robots, the ev3dev framework [ev] allows to address sensors and actors via the file system. ev3dev is an open source OS for LEGO EV3 based on Debian Linux. It provides a low-level driver framework for controlling EV3 brick devices, sensors, and motors. Usually, ev3dev is run on the EV3 brick by booting from an SD card containing the ev3dev image. SWI-Prolog is then readily available for ev3dev using the built-in package manager.

In this paper, we describe how ev3dev can be used to control EV3 robots by Prolog programs. Our API consists of three layers. In the driver framework (ev3dev predicates), ev3dev file access operations are wrapped into Prolog predicates. Predicates in the second layer (low level predicates) are used to read and control single sensors and actors. They do not access files directly but use ev3dev predicates from the driver framework. On top of this basic library we define higher level abstractions in the upper layer of our API to provide an application programming interface for configuring and controlling robots.



To control an actor or read a value from a sensor, ev3dev provides a number of virtual files that can be read from and written to, as shown in the directory listing for a motor.

```
robot@ev3dev:~$ ls -l /sys/class/tacho-motor/motor0/
-r--r--r-- 1 root ev3dev 4096 Jun 28 13:42 address
--w--w---- 1 root ev3dev 4096 Jun 28 13:42 command
-r--r--r-- 1 root ev3dev 4096 Jun 28 13:42 commands
-r--r--r-- 1 root ev3dev 4096 Jun 28 13:42 max_speed
-r--r--r-- 1 root ev3dev 4096 Jun 28 13:42 speed
-rw-rw-r-- 1 root ev3dev 4096 Jun 28 13:42 speed_sp
-r--r--r-- 1 root ev3dev 4096 Jun 28 13:42 state
```

To run a motor, `run-forever` is written in its `command` file. Which commands are actually supported by the motor can be read from the `commands` file. The motor is stopped by writing `stop` in the `command` file. To facilitate this, the driver framework forms a thin wrapper around the provided accessor files and ev3dev module structure. The program below contains only low level ev3dev predicates.

```
command(Port, Command) :-
    command_file(Port, File), file_write(File, Command).
command_file(Port, File) :-
    tacho_motor(Port, _, Basepath),
    atomic_concat(Basepath, '/command', File).
tacho_motor(Port, Type, Path) :-
    subsystem_detect(Port, Type, Path, '/sys/class/tacho-motor/motor*/').
subsystem_detect(Port, Type, Path, Prefix) :-
    expand_file_name(Prefix, Paths), member(Path, Paths),
    atomic_concat(Path, '/address', AddressFile),
    atomic_concat(Path, '/driver_name', DriverFile),
    file_read(AddressFile, Port), file_read(DriverFile, Type).
```

If the parameters `Port`, `Type`, and `Path` are not bound, the predicate `subsystem_detect/4` will bind them as per the device detection by ev3dev. Consequently, if there is no tacho motor attached to the given port, the predicate `tacho_motor/3` fails. This can be used to check whether the implementation of the program fits the physical configuration of the robot.

3 Motor and Sensor Control in ev3dev-prolog

3.1 Reading Sensor Values

All low level predicates presented in Section 2 correspond to ev3dev interface files. Tasks like “get sensor value at port *X*” or “stop motor at port *X*” are combinations of multiple basic actions, each represented by a low level predicate defined in Section 2. ev3dev-prolog provides predicates for this type of combined actions.

To read a sensor value, it is necessary to check whether there is actually a sensor of the supported type attached to the specified port, set the operating mode of the sensor to the mode corresponding to that sensor value, and then read a specific file containing the sensor value that corresponds to the operating mode.

For instance, the predicate to read the current value of a color sensor is implemented as

```
col_ambient(Port, Val) :-
    lego_sensor(Port, 'lego-ev3-color'),
    mode(Port, 'COL-AMBIENT'),
    value(Port, 0, Val).
```

3.2 Motor Control

To run a motor for a given number of rotations with a given percentwise speed, that speed needs to be converted to internal motor speeds, the motor needs to be set running and the predicate has to delay its complete evaluation until the motor has stopped. If the evaluation of the predicate would be finished as soon as the motor starts running, following actions would interfere with unfinished actions. This is a problem in many robot control languages. In some of them, even the semantics of related language constructs remains unclear. ev3dev-prolog provides combined predicates to control motors, for instance

```
motor_run(Motor, Speed, Angle) :-
   speed_adjust(Speed, Motor, CSpeed),  speed_sp(Motor, CSpeed),
   position_sp(Motor, Angle),  command(Motor, 'run-to-rel-pos'),
   motor_wait_while(Motor, 'running').
speed_adjust(PercentVal, MotorPort, Speed) :-
   max_speed(MotorPort, MaxSpeed),
   Speed is floor(PercentVal / 100.0 * MaxSpeed).
```

The predicate `motor_wait_while/2` allows the interpreter to stall execution of further actions and evaluations until the motor reaches a state that implies that the action is finished. Possible motor states are running, stalled, or holding.

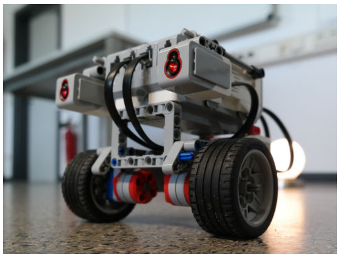
```
motor_wait_while(Motor, State) :-
 tacho_motor(Motor), repeat,
 motor_states(Motor, States), \+ memberchk(State, States),!.
```

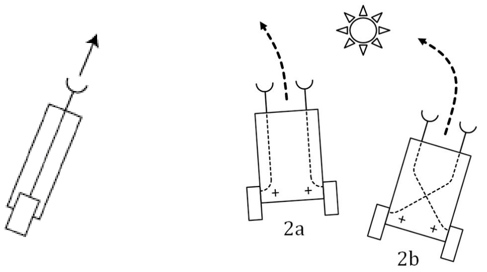
3.3 Application of the Low Level API: Braitenberg Vehicles

The predicates presented in the previous sections are sufficient to implement simple robot behaviour in intuitively understandable predicates. In our introductory robotics courses, we use two-wheeled Braitenberg vehicles (see [St85], where also the schematics are from). In Braitenberg vehicles, the connection between cognition and action is simple and direct. Every movement of the vehicle is an immediate reaction to the output value of a sensor.

There is no intelligence involved. The values of (light or ultrasonic) sensors directly determine the speed of the wheel motors. Because this simple construction can generate surprisingly complex behaviour, experiments with Braitenberg vehicles are suitable to draw interest in robotics.

Our EV3 Braitenberg vehicle and schematics of the movement of Braitenberg vehicles.





Braitenberg vehicle 1

Higher light intensity produces faster movement. Less light produces slower movement. Darkness produces standstill. The control of this robot is implemented by the following predicate. Recursively, the intensity received from one light sensor is applied to both motors.

```
braitenberg1a :-
    col_ambient(_, Light), forall(tacho_motor(M), motor_run(M, Light)),
    !, braitenberg1a.
```

Braitenberg vehicle 2

The left light sensor corresponds to the left motor's rotation and the right light sensor corresponds to the right motor's rotation. The vehicle is turning away from the light because the motor that is on the same side as the light source runs faster than the other one.

```
braitenberg2 :-
    col_ambient(in2, LightR), motor_run(outB, LightR),
    col_ambient(in3, LightL), motor_run(outC, LightL),
    !, braitenberg2.
```

By exchanging the ports for both sensors or both motors, either in software or physically, the vehicle changes behaviour by turning towards the light since the motor on the darker side rotates faster than the one on the light side. Using a mobile light source like a torch, funny experiments are possible, since students can be chased by the robot or lead the robot around the room.

4 Robot Control in ev3dev-prolog

4.1 Robot Configuration

Robots have certain physical properties. Basic robots usually have two motors with wheels on either side using so called “tank controls” that can be controlled independently. To use the motors’ odometry to move a specific distance or turn a specific angles, the robot needs information about the sizes of the attached wheels as well as the distance of both wheels.

Evaluating the `set_robot/4` predicate, the interpreter not only checks that all variables are instantiated and that there are motors attached at the specified ports. It also checks that both motors are of the same type. Different motor types have different physical properties, like acceleration and maximum speed. Therefore, the combination of different motor types to a tank control is usually considered as invalid. In the following predicate, the matching type of both motors is guaranteed by the binding of the variable `Type`.

```
set_robot(WheelDiameter, AxleLength, LeftMotorPort, RightMotorPort) :-
    nonvar(WheelDiameter), nonvar(AxleLength),
    nonvar(LeftMotorPort), nonvar(RightMotorPort),
    tacho_motor(LeftMotorPort, Type),
    tacho_motor(RightMotorPort, Type),
    retractall( robot(_, _, _, _) ),
    asserta(robot(WheelDiameter, AxleLength, LeftMotorPort, RightMotorPort)).
```

4.2 Robot Movement

As mentioned before, motor actions that take some time to execute should block the corresponding predicate from returning before the motor action is finished. This is critical because both motors have to run simultaneously to move the robot straight forward. Therefore, each motor is started in a separate thread (using `thread_create/3`) to allow for parallel evaluation and execution. Since the predicate is blocking, once both threads are joined again, the motors have stopped and the given command was finished.

```
go(Speed) :-
  Speed \= 0, robot(_, _, LM, RM),
  motor_run(LM, Speed), motor_run(RM, Speed).
go(Speed, Angle) :-
  Speed \= 0, Angle \= 0,
  robot(_, _, LM, RM),
  thread_create(motor_run(LM, Speed, Angle), Id1, []),
  thread_create(motor_run(RM, Speed, Angle), Id2, []),
  thread_join(Id1, true), thread_join(Id2, true),!.
go_cm(Speed, Distance) :-
  robot(WD, _, _, _),
  Angle is round((Distance*360)/(pi*WD)),
  go(Speed,Angle).
```

```
stop :-
  robot(_, _, LM, RM), motor_stop(LM), motor_stop(RM).
```

Prolog's evaluation strategy (SLD resolution) allows to combine multiple implementations for robot actions, depending on the components of the robot. The robot might have a gyroscopic sensor detecting rotation in the plane. Movements controlled by sensed rotation is usually more accurate than purely relying on internal motor measurements. Using Prolog's evaluation strategy, we can use this sensor to make the robot turn a certain angle. If no gyroscopic sensor is connected to the robot, a fallback to odometry is used.

```
turn(Speed, Angle) :-
    gyro_sensor(Port), NSpeed is -Speed,
    stop, gyro_reset(Port),
    repeat, gyro_ang(Port, ReadAngle),
    Diff is Angle - ReadAngle,
    ( % approach target angle slowly
      (Diff = 0, stop,!);
      (Diff > 0, turn(min(Speed, Diff / 4)), fail);
      (Diff < 0, turn(max(NSpeed, Diff / 4)), fail)
    ).
turn(Speed, Angle) :-
    robot(WD, AL, LM, RM), MAngle is round(AL/WD*Angle),
    NegMAngle is -MAngle,
    thread_create(motor_run(LM, Speed, MAngle), Id1, []),
    thread_create(motor_run(RM, Speed, NegMAngle), Id2, []),
    thread_join(Id1, true), thread_join(Id2, true),!.
```

For continuously turning the robot in place without a target angle, ev3dev-prolog contains the predicate `turn/1` that runs the motors counter-rotating.

```
turn(Speed) :-
    robot(_, _, LM, RM), NSpeed is -Speed,
    motor_run(LM, Speed), motor_run(RM, NSpeed).
```

4.3 Application of the High Level API: Obstacle Avoidance

An application using these higher level commands is a robot that detects and avoids obstacles. The robot should go forward until it spots an obstacle and then turn away from it, continuing to go forward after the turn.

```
start :-
    set_robot(5.6, 10.6, outB, outC),
    obstacle_avoidance.
obstacle_avoidance :-
    ((us_dist_cm(_, Dist), Dist > 50, go(20));
    turn(20, 90)),!,
    obstacle_avoidance.
```

Note that this implementation of `obstacle_avoidance/0` is abstract and does not use any specific motor port or physical aspect of the actual robot. The correct movement is controlled by the robot configuration. The sensor measurement works without providing the actual port of the ultrasonic sensor. As long as there is an ultrasonic sensor attached, it will be auto-detected and the measurement taken. In this application we only assume that there is one sensor attached to the robot in a useful orientation. In applications that uses multiple sensors, they can be distinguished by explicitly stating the port each sensor is plugged into.

5 Basic Robot Commands in ev3dev-prolog

Actions of mobile Robots are usually controlled by commands like

go (a given distance) at a given speed
turn (a given angle) at a given speed
stop

This small collection of commands for basic movements is common to most control languages for mobile robots. We present an implementation of these basic commands in ev3dev-prolog. It allows to generate complex movements of two-wheeled LEGO EV3 robots and other user-defined robot configurations.

High level API

Predicate	**Parameters**	**Effect**
`set_robot/4`	Wheel diameter (in cm) Axle length (in cm) Port of left motor Port of right motor	Define the specific parameters of the robot model
`stop/0`		Stop all motors of the robot
`go/1` `go/2`	Speed (in %) Angle (in degrees)	Turn the robot's motors continuously or a certain angle
`go_cm/2`	Speed (in %) Distance (in cm)	Move the robot a certain distance
`turn/1` `turn/2`	Speed (in %) Angle (in degrees)	Turn the robot continuously or a certain angle

To control robots with grippers, for instance, ev3dev-prolog can be extended by commands to open and close the gripper.

The commands from the high level API are translated to Prolog predicates that trigger low level motor and sensor actions as defined in Section 2. If the execution of a move is impossible, the according predicate fails.

To produce correct movements, the translation depends on the specifics of the robot model like the diameter of its wheels and the distance between them. The configuration of a robot with two motors, each of them connected to one wheel, is set by a predicate `set_robot/4` with parameters WheelDiameter, AxleLength, LeftMotorPort, RightMotorPort. By the Prolog evaluation strategy, some system checks are easyly integrated in this specification predicate. For instance, if there is no motor connected to `outB`, the predicate `set_robot(5, 10, outA, outB)` will fail.

This high level part of ev3dev-prolog hides the physical configuration of the robot. To the user, the configuration of the robot is only accessible via the predicate `set_robot/4`. In more complex robot models, it is necessary to directly access sensors and motors. For example grippers, as well as many sensors with myriad of ways they can be attached to or removed from a robot. In the robot construction phase, it is not always reasonable to define a robot configuration for every special robot. For direct access to motors and sensors, our API contains a set of lower level predicates. These predicates can be parameterized by the physical port of the sensor or actor.

Low level API

Predicate	Parameters	Effect
`us_dist_cm/2`	Sensor port (`in1`, `in2`, `in3`, or `in4`)	Read an ultrasonic sensor in distance and cm mode
`col_ambient/2`		Read a light sensor in ambient light mode
`col_reflect/2`		Read a light sensor in reflected light mode
`gyro_rate/2`	Sensor value	Read a gyroscopic sensor in rate of change mode
`motor_stop/1`	Motor port (`outA`, `outB`, `outC`, or `outD`)	Stop a motor
`motor_run/2`	Motor port Speed (in %)	Run a motor continuously or for a certain angle
`motor_run/3`	Angle (in degrees)	

Currently, ev3dev-prolog supports the operating modes of the light sensor, ultrasonic sensor, gyroscopic sensor, and touch sensor. Using our base predicates for file access, this can easily be extended to other sensors, like, temperature sensors. ev3dev-prolog can control all motors supported by the tacho motor system, i.e., large and medium EV3 motors and the older NXT motors.

6 Moving robots in unknown environments

In this section, we demonstrate how to use ev3dev-prolog to move robots in unknown environments. Currently, the environment is represented by a rectangular 2D grid (see [La06]). Cells of this grid can be blocked by obstacles.

The pose of a robot consists of position and direction of its movement. The robot knows the coordinates of its goal position relatively to its starting pose and shall move to this position. At the beginning, the robot knows nothing about obstacles on its way to the goal.

In AI lectures, we are using these robots to examine and compare different planning algorithms like A^* search or bidirectional planning. Therefore, the path planner is a black box in the following implementation.

Using the planner, the robot generates a plan to its goal position and starts to move according to this plan. If the robot discovers an obstacle that prevents to execute a planned movement, this new information about the environment is included into the internal knowledge base and replanning starts from the robot's current pose.

```
plan_and_go_to([TX, TY], [DX, DY]) :-
    repeat, state_position([SX, SY], [SDX, SDY]),
    findnsols(1, Plan, plan(Plan, state([SX, SY], [SDX, SDY]),
                              state([TX, TY], [DX, DY])), _),
    execute_plan(Plan),!.
execute_plan([fin]).
execute_plan([Move|Tail]) :- !, Move, execute_plan(Tail).
```

Note that only one plan is created. If the execution of this plan fails in a pose different from the starting pose, no alternative plan from the original position would be applicable. Therefore any backtracking has to return to the planning phase to create a new plan from the current pose with the extended knowledge about the environment.

High level ev3dev-prolog predicates allow to create planning predicates that on evaluation change the global state (`state_position/2`) of the robot. During the planning phase we assume that every move is possible and its effects are simulated.

```
plan_fits([step(fin, TargetField, TargetDirection)],
          state(TargetField, TargetDirection)).
plan_fits([step(move_if_free, [FX, FY], [DX, DY])|Tail], Goal) :-
    TX is FX + DX, TY is FY + DY, \+ obstructed([TX, TY]),
    Tail = [step(_, [TX, TY], [DX, DY])|_], plan_fits(Tail, Goal).
```

Even if a plan is found, it might fail completely during the execution phase due to obstacles blocking parts of the planned path.

Obstacles are detected by an ultrasonic sensor as explained in the obstacle avoidance example in Section 4.3. The execution phase gives feedback to the planning phase by asserting predicates that represent global obstacles.

In the implementation below, obstacles are static and are not retracted later. Dynamic obstacles can be handled by marking obstacles with some additional data on when they can be retracted and how their influence on the planing phase may change over time.

```
free :- us_dist_cm(_, Distance),
        ( Distance > 20; state_position([X,Y],[DX, DY]),
          TX is X+DX, TY is Y+DY,
          asserta(obstructed([TX, TY])), fail ).
```

If only one ultrasonic sensor is connected to the robot, it is not necessary to define the port of this sensor. Hence the user does not even need to know the port where the sensor is connected. Even reconnection of the sensor to another port does not require any change in the program. If no ultrasonic sensor is connected to the robot, the predicate `us_dist_cm(_, Distance)` fails.

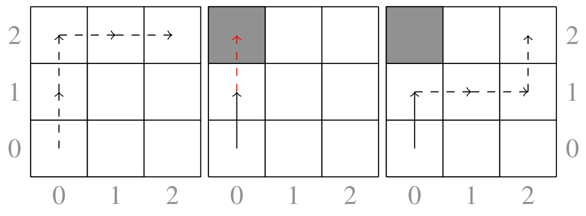
This is an advantage compared to common control languages for Lego robots like the graphical EV3 programming interface, LeJos, and Nalepa's Prolog API [Na08].

Changes in the robot pose are adjusted by retraction of the previous state and assertion of the current state.

```
move :- state_position([X, Y], [DX, DY]),
        go_cm(10, 20), retract(state_position([X, Y], [DX, DY])),
        TX is X+DX, TY is Y+DY,
        asserta(state_position([TX, TY], [DX, DY])).
move_if_free :- free, move.
```

Let, for example, the robot move from $(0,0)$ to $(2,2)$ on an empty grid. The first plan the planner proposes is: moving forward twice, turning right, and moving forward twice again. On execution after the first move – while the robot is at $(1,0)$ – the second move fails if an obstacle is detected at $(2,0)$.

Dynamic planning



Then the planning phase starts anew and the new plan might be: a right turn, twice forward to $(1,2)$ and a left turn followed by a final move forward. The execution of this plan might succeed or not. If another obstacle is detected on this path, it is added to the global obstacle database and another plan is created from the new position accounting for the new obstacle.

Note that the execution of plans is independent of the implementation of the planner. Before moving to the next cell, the robot checks whether the move can be performed and stops the execution of the current plan if not. Therefore it is not possible to execute a plan that, for example, crashes the robot into a wall. The robot will not enter the blocked cell. If necessary, replanning is performed until a plan is found that can be executed completely or, according to the internal knowledge, the goal position is not accessible from the current pose of the robot.

7 Discussion

The original contribution of this paper is ev3dev-prolog – an extendable Prolog API for LEGO EV3 robots. We demonstrate the application of this API in several robotics experiments used in lectures on robotics, AI, and logic programming.

The reactive behaviour of Prolog programs is interesting in itself. Examining the programs and the resulting behaviour helps to understand the backtrackable reasoning process in Prolog interpreters. This is one reason, why we want to use the presented approach in our lectures on logic programming.

Advanced experiments involving planning problems and other tasks that can be solved by Prolog programs demonstrate the power of the logic programming paradigm in robotics. In [LP00] and [Ca09], advanced AI approaches are documented for predecessors of the LEGO MINDSTORMS EV3 platform. Because of the restricted computing power in those earlier versions, plans were mostly generated outside the robot and then transferred to the robot for execution. The enhanced computing power of the EV3 robot allows computation and modification of plans as well as their execution directly in the robot.

As presented in Section 5, ev3dev-prolog allows to implement complex robot behaviours concisely in a declarative way. ev3dev-prolog provides a convenient and easily accessible way to demonstrate the power of logic programming and AI in robot control. As shown in the example in Section 6, ev3dev-prolog can be used to demonstrate and compare rule based AI techniques for knowledge representation and processing with LEGO EV3 robots. This enables an easy integration of motivating experiments and competitions in in lab sessions on robotics, AI and logic programming.

An extension of ev3dev-prolog by bindings to other sensors and actors, like, display, speaker, and keypad LEDs on the EV3 brick could be useful. ev3dev-prolog should also be improved by predicates to move robots along given trajectories like curves with parameterizable radii. An extension of our path planning example from 2D grids to arbitrary 2D environments will be the next step into more complex navigation experiments.

References

[Ca09] Caldiran, O.; Haspalamutgil, K.; Ok, A.; Palaz, C.; Erdem, E.; Patoglu, V.: Bridging the Gap between High-Level Reasoning and Low-Level Control. In: LPNMR. Vol. 5753. Lecture Notes in Computer Science, Springer, pp. 342–354, 2009.

[ev] ev3dev: ev3dev, `http://www.ev3dev.org`, Accessed: 2019-07-01.

[HH02] Hanus, M.; Höppner, K.: Programming Autonomous Robots in Curry. Electr. Notes Theor. Comput. Sci. 76/, pp. 178–196, 2002.

[La06] LaValle, S. M.: Planning algorithms. Cambridge University Press, 2006.

[Le] Lego: LEGO MINDSTORMS EV3, `https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313`, Accessed: 2019-07-01.

[Le97] Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; Scherl, R. B.: GOLOG: A Logic Programming Language for Dynamic Domains. J. Log. Program. 31/1-3, pp. 59–83, 1997.

[LP00] Levesque, H. J.; Pagnucco, M.: Legolog: Inexpensive experiments in cognitive robotics. In: In Proc. of CogRob2000. 2000.

[Lu16] Lu, W.: Beginning Robotics Programming in Java with LEGO Mindstorms. Apress, Berkely, CA, USA, 2016.

[Na08] Nalepa, G. J.: Prototype Prolog API for Mindstorms NXT. In: KI. Vol. 5243. Lecture Notes in Computer Science, Springer, pp. 393–394, 2008.

[Po95] Poole, D.: Logic Programming for Robot Control. In: IJCAI. Morgan Kaufmann, pp. 150–157, 1995.

[RN95] Russell, S. J.; Norvig, P.: Artificial intelligence - a modern approach: the intelligent agent book. Prentice Hall, 1995.

[St85] Stefik, M.: V. Braitenberg, Vehicles: Experiments in Synthetic Psychology. Artif. Intell. 27/2, pp. 246–248, 1985.