A Novel Development Paradigm for Event-based Applications

Jörg Roth
Faculty of Computer Science
Nuremberg Institute of Technology
Nuremberg, Germany
Joerg.Roth@th-nuernberg.de

Abstract—**Application development for highly dynamic and unpredictable scenarios still is a difficult and time-consuming task. This in particular is a problem in the area of robotics: if a certain task should be executed, many asynchronous events can prevent a plan from its termination. In contrast to applications in e.g. office scenarios, interruption as a result of unexpected conditions is the normal case. In this paper we present a novel development approach. It supports graphical programming and allows a developer to easily formulate complex tasks. The model covers both the flow of actions and the flow of data in a single representation. In addition, it simplifies the modeling of interruptions and asynchronous events. We implemented and tested the approach with our Carbot platform.**

Keywords—Event-based Applications, Robotics, Graphical Programming

I. Introduction

Mobile robots promise a high potential for the future in the area of assisted living, health services, industrial production, entertainment and housekeeping. A long time research in this area required highly specialized knowledge about electronics and mechanical construction, but recent developments in hardware, mechanics, sensors and software enable to built robotics platforms not only for experts. Besides expensive and highly specialized robots (e.g. for rescue tasks), we nowadays can think about low-cost robots with awesome capabilities. This expands the opportunity to construct and program robots for new groups such as students or even interested adolescents with low programming skills. The enormous success of the Mindstorms series in school and academic education is an example of this development.

This change, however, causes a problem. If the mechanical construction and hardware is completed, the programming is still a difficult task. This is because robots interact in highly dynamic, only partly known environments. Asynchronous events are the usual case. This makes an apparently simply task such as 'drive to the centre of the room' very difficult. During driving, an obstacle may appear. While backing up to circumnavigate the obstacle, the robot may drive against a wall etc. To formulate the corresponding rules in a traditional programming language produces code that is difficult to read and to maintain. In particular the large number of simultaneous checks is a high burden for traditional programming: simultaneous execution requires a) a high level of structured thinking and abstraction; b) the knowledge of facilities to control concurrency (e.g. monitors or semaphores) and c) effort to deal with all kinds of concurrency issues such as critical sections or race conditions.

In this paper we introduced the new concept of *Action Flow Plans*. It is designed according to the following goals:

- A plan both incorporates normal plan execution as well as how to deal with unexpected interruption in a single representation.
- It supports graphical programming. Programming is mainly performed by drag and drop.
- It is suitable for a wide range of programming skills from experts to interested people with only minor programming skills.
- It supports a wide range of applications ranging from basic tasks (e.g. 'find the infrared source') to very complex tasks ('navigate to room xy').
- A developer is able to develop even low-level code or integrate code from outside the environment. Foreign code may access plan code and data entries. In turn, plan code may call code outside the platform.

We fully implemented a complete tool chain including a graphical editor and a runtime environment for our *Carbot* platform.

II. Related Work

The most common development style for code in our intended scenarios still is based on traditional development paradigms, either non-object-oriented (e.g. C [5]) or object-oriented (e.g. Java [1]). As such, this is not a problem as appropriate tool environments give the developer full control over all sensors and motors. However, a developer is not forced to structure the code to reflect blocks of actions such as 'drive forward until you reach an obstacle'. I.e. a complex action may be distributed among different code locations. It also depends on the developer to document the dynamics of mutual interruptions by asynchronous events. As a consequence, resulting code *may* explain the overall behavior to other developers, but it also can be very difficult to read and maintain it.

A popular programming environment that tries to address these issues is NXT-G [7] (or its successor EV3-G). It is based on LabView [3] and forms the basic software-development tool for the Mindstorms series. NXT-G programs are implicitly parallel. The developer can start multiple execution flows that are processed in thread manner. Programming is done graphically: the developer places blocks on one of the sequence bars that represent a thread.

NXT-G blocks provide very low-level functions such as measure the ultrasonic value or turn the motor 3 times. Data between blocks is passed in a data flow manner, i.e. a distance measurement can directly be passed to a motor input, to e.g. always keep a certain distance to an object. There exist blocks for loops and decisions (similar to *for-i* or *if-then-else*). It is easy for a developer to formulate tasks that are built up by periodic actions. However, complex tasks with different stages lead to very complex plans. Asynchronous events are difficult to handle: it is not intended for one thread to interrupt another thread, thus asynchronous events have to be signaled by data. This however is not an appropriate way to propagate events in many cases. Moreover, as there is no notion of plan states, it is difficult for a thread to conditionally interrupt only, if the robot currently is in a certain state.

Even though developers can create their first program in a minute, some complex programs are either difficult or even virtually impossible. It is not intended for a developer to integrate own code into blocks, call plans from own code or to pass own data to NXT-G programs from outside.

A more high-level approach to structure complex tasks provides the *subsumption* architecture [4][8]. It assumes that complex tasks can be built up by independent *behaviors*. Behaviors are tasks like 'avoid an obstacle' or 'find the ball'. At runtime, behaviors try to get control. If multiple behaviors can be active, a selection mechanism makes a decision. Subsumption architectures can easily be modeled using an object-oriented class system. E.g. in *Lejos* [1] a class **`Arbitrator`** represents a container of behaviors. It selects an active behavior at runtime. A class **`Behavior`** provides three methods: **`takeControl`**→**`boolean`** indicates, whether a behavior is able to take control of the robot; **`action`** executes the behavior's action; and **`suppress`** stops the action whenever the arbitrator decides to shift control to another behavior.

Behaviors provide a basic structure of actions in a complex task. However, some drawbacks: first, behaviors have to control complex plans themselves (usually in **`takeControl`**). E.g., if a list of behaviors should be executed consecutively, the sequence schema is coded in a distributed manner in the respective behaviors. Second, there is no explicit notion of communication between behaviors as they are considered as independent tasks. Third, behaviors cannot directly interrupt a specific other behavior. E.g. a behavior 'detect obstacle' should be able to interrupt 'forward driving' but not 'look around'. As the only instance that may interrupt a behavior is the arbitrator, a behavior itself cannot explicitly interrupt another behavior.

Considering existing work, we can formulate our goals as follows:

- We want to compose complex tasks from high-level components. We consider the blocks such as in NXT-G as too low-level to create complex tasks.
- We want to explicit model both flow of actions and flow of data in the same plan.
- We want to explicitly model interruption of activities, as we consider interruption as a result of an asynchronous event as a normal case.

We further want to directly support graphical programming as we assume this reduces the barrier for developers.

III. ACTION FLOW PLANS

Our approach of Action Flow Plans enables a developer to quickly formulate complex plans that can execute pre-defined sequences and react to asynchronous events. Before we describe the Action Flow Plans in more detail, fig. 1 (left) presents the different development roles in our approach.

Plans are formulated with the help of predefined *actor* and *supervisor* templates (see below). A *toolbox developer* thus first has to specify suitable templates for the intended runtime environment. The templates are formulated in an object-oriented manner and contain the actual runtime code, but also specifications for the usage in our graphical development environment (e.g. icons and help texts).

The *application developer* plays the major role on our approach. If once a toolbox is set up, we only need this developer to generate an application. The application developer takes templates and creates an Action Flow Plan with the help of the graphical development environment (fig. 1, right).

The graphical development tool allows the developer to take templates from the left row and to mount a plan by drag and drop. In addition, the tool contains a function to perform a consistency check. If a plain is valid, the developer can generate runtime code from the plan. An execution plan has a straightforward relationship to the generated source code, where templates correspond to classes in an object-oriented programming language and plan elements correspond to their instances.

Finally, a runtime integrator can take this code and merge it with arbitrary other code. This last step allows to even use low-level functions of the runtime system. Our approach both allows to call code from within our action flow plans, but also to execute plans and sub-plans from within other code of the system.

In small projects a single developer can play all three roles. Even though the developer then has to deal with platform-related code, the approach still benefits from a clear description of plans that easily can be maintained independently from low-level code.

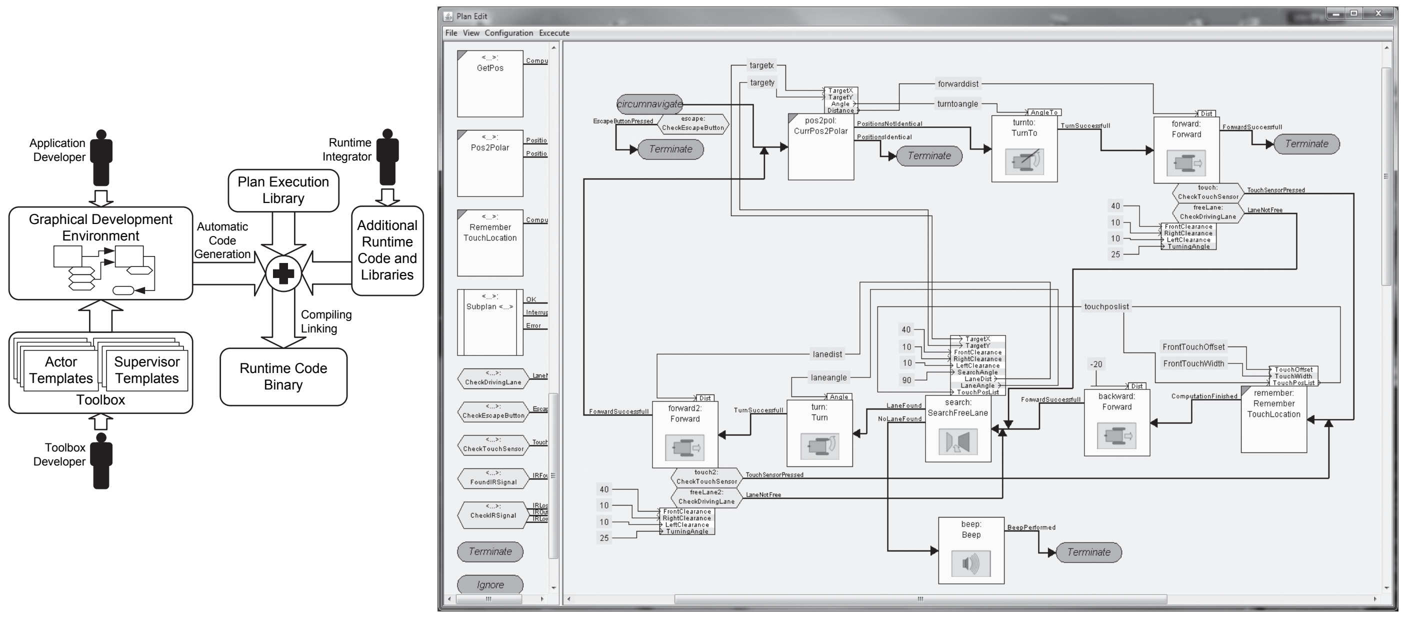

Fig. 1. Different development roles (left), the graphical development environment (right)

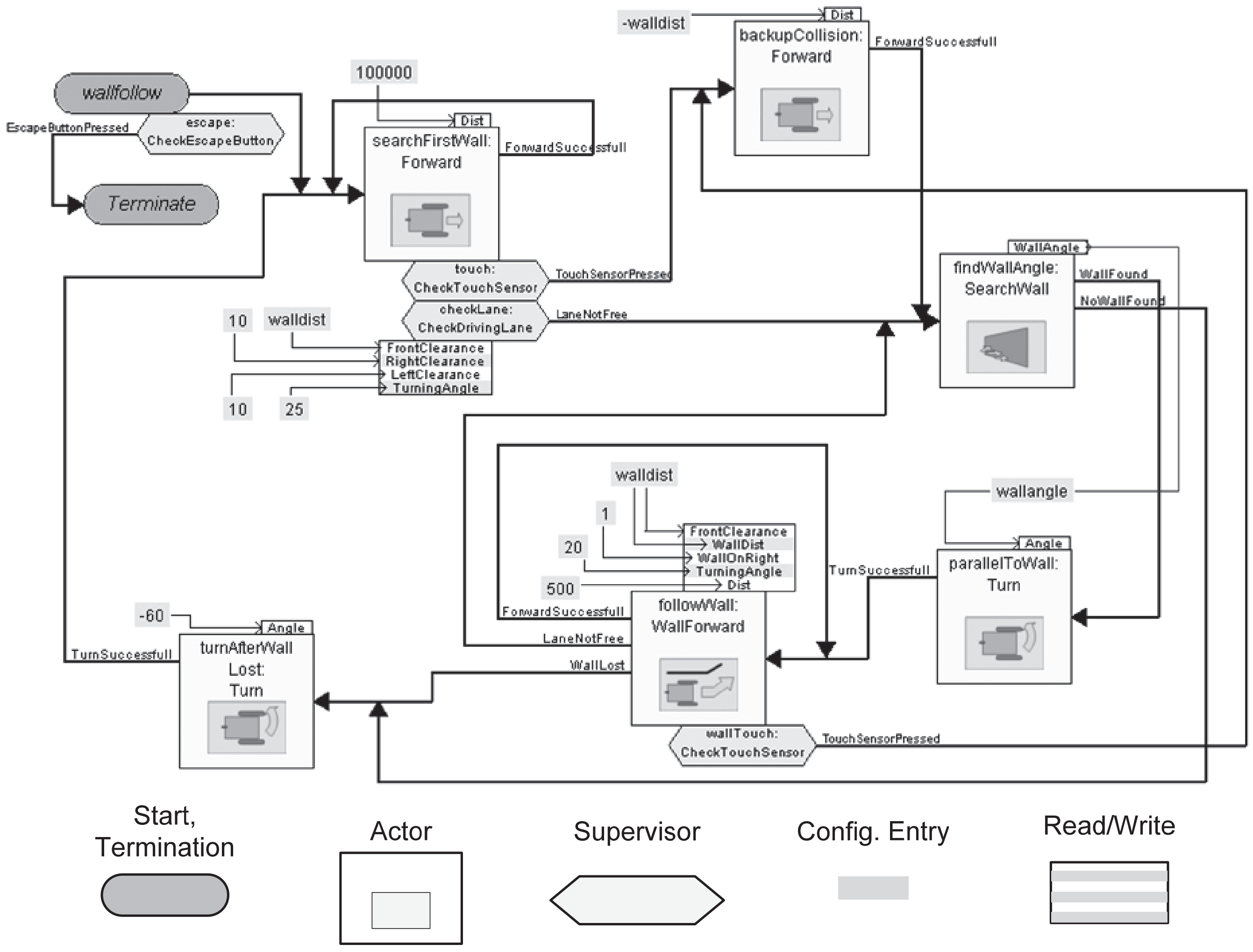

Fig. 2. An Action Flow Plan for 'wall following'

A. *The Action Flow Plan Model*

Our model contains the following components:

- A *plan* contains a sequence of actions for a certain task. Larger plans may be organized with the help of *sub-plans*.
- An *actor* describes a complete action such as *turn 90^o* or *drive forward 100 cm*. These actions may have different result *states* and may be interrupted during execution. If an actor only modifies internal variables, we call it *computation*.
- A *supervisor* contains code to check conditions that may interrupt an action. E.g. a supervisor may check for obstacles in driving direction and is able to interrupt forward driving. Supervisors are attached to actors or may be global.
- The *configuration* is used to store internal states and contains a pool of named variables. Actors and supervisors may read from the configuration to parameterize their execution and may write to the configuration to specify the result of an action.

The model supports two types of connections: flow of actions (thick lines) and flow of data (thin lines). Both connections are directed and either indicate a timely sequence or the writing direction of data.

We use an example to describe the interaction of actors, supervisors and configuration. We want to create a plan for the task: *find the next wall, then drive parallel to the wall until the user presses an escape button*. We can sketch this task as follows:

- Drive forward until you reach a wall.
- Measure the angle to the wall, then turn the robot until the wall is on the right side, parallel to the driving direction.
- Drive forward; permanently measure the distance on the right and try to keep the distance.

As such, this plan is simple. However, some asynchronous events can violate the plan execution:

- Whenever the robot drives forward, an unrecognized obstacle can appear (detected, e.g., by a tactile sensor). Then, the robot should stop, backup and drive around the obstacle.
- The robot tries to keep a certain distance to the wall. This may fail, if the wall ends. In this case, the robot should search the wall again.
- The user may press a button to stop the execution at any time.

Fig. 2 presents the resulting plan. The definition of actors and supervisors is obvious: actors model the steps of the plan during plan execution; supervisors model the asynchronous events that prevent the plan from normal execution.

The configuration contains constant values that parameterize the actors. The value **`walldist`** is a variable that can be globally set (especially by the application that starts the plan execution). Here, the value defines the fix distance to the next wall, the robot tries to keep. A second value, **`wallangle`** is both read and written: the actor **`findWallAngle`** measures the distance to the next obstacle at different points. If all points reside on a straight line, the next state is **`WallFound`** and the angle of these points is stored in the variable **`wallangle`**. This then is read by the actor **`parallelToWall`** that turns the robot to be parallel to the wall.

Some supervisors check for interruption during an actor execution. E.g. when driving forward to find a wall, two supervisors permanently check, whether the driving direction is free from obstacles. If not, the actor is interrupted and another actor tries to deal with the new situation.

Some actors may go to different states, even though they terminate without interruptions by supervisors. E.g., **`findWallAngle`**, may find a wall angle or not. If not, the robot turns, drives forward and again tries to find a wall. The actor **`followWall`** may 'loose' the current wall, then it goes into the state **`WallLost`**.

B. *The Model in Detail*

Fig. 3 shows all elements of Action Flow Plans.

During plan execution, there always is exactly *one* actor active. The first active actor is marked by a start tag that holds the plan name (e.g. 'Plan' in fig. 3).

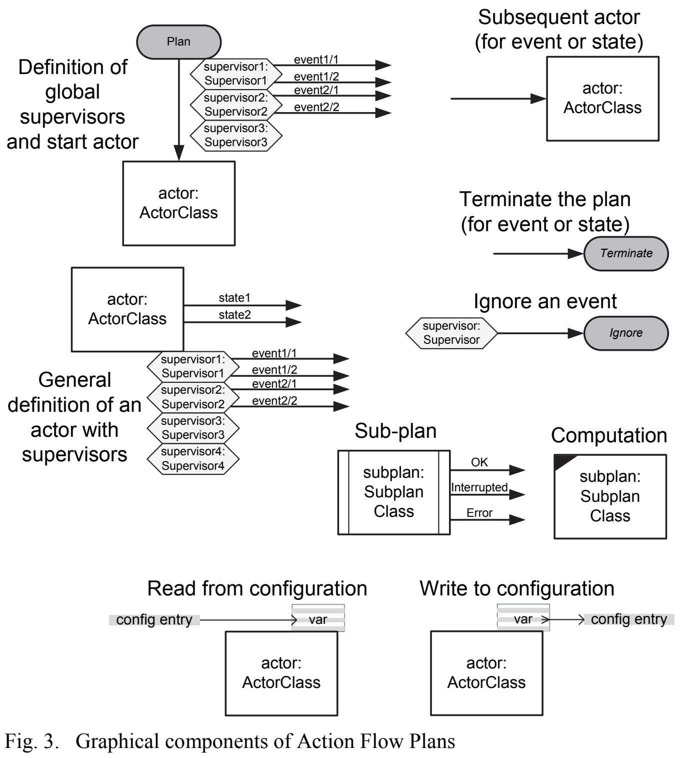

Fig. 3. Graphical components of Action Flow Plans

The start tag may also hold some supervisors – the *global supervisors*. These are active all the time and may interrupt any active actor. A typical global supervisor is the check for the stop key in order to terminate on user request.

If an active state terminates normally, it reaches one of multiple *states*. A state either is connected to a subsequent actor or to the tag *Terminate*. An actor is interrupted if one of the global supervisors or one of the supervisors assigned to this actor detects an *event*. The corresponding event is connected to a) a subsequent actor b) tag *Terminate* or c) tag *Ignore*. The latter is only reasonable for supervisors that can check multiple events, of which some should not interrupt an actor.

Any plan can become a *sub-plan* for a larger plan. For the larger plan, a sub-plan is considered as a normal actor with one difference – the states of a sub-plan are fixed: `OK` (terminated without interruption), `Interrupted` (terminated by supervisor) and `Error` (internal exception occurred).

Computations are special actors that only modify the configuration. They have a nature of a function call that does not cause a side effect. Computation actors have a special graphical symbol, but their major meaning is to clarify a plan. It will not technically be checked, if the corresponding template code causes side effects. A computation may not have any supervisor, as it is expected to immediately terminate. A typical example for a computation is the component that controls a *for-i* loop: it reads the counter value, stores its increment and goes to one of two states: one to again go into the loop and one for terminating the loop.

Actors and supervisors may read from and write to configuration entries. This mechanism can be viewed as parameter lists of method calls. Configuration entries have a type that is a native type (e.g. `int`, `float`, `String`) or `Object`. `Object` types can be viewed as pointer, where only a certain actor knows the internal structure. This allows an actor to store even complex data in the configuration.

The variable type is defined by the actor or supervisor. If the application developer connects a configuration entry to a variable with different type, a type conversion is tried.

Usually the configuration entry is defined by a single string that can be viewed as a global variable name. Sometimes, the values of configuration entries should be modified according to simple formulas, e.g. assign `2*dist+5` to input variable `dist2` of an actor. Even though this problem could be solved by computations, this would lead to a large number of different computations that only execute simple formulas. Thus, we decided to formulate simple formulas in configuration entries. These can contain:

- numerical and textual constants;
- basic arithmetic operations and string concatenation;
- brackets that control the ordering of evaluation.

Configuration entries that contain formulas cannot be written. If such a formula is getting evaluated, also type conversion is tried, if input from different types is received.

Note that configuration entries are also a tool for a runtime integrator to establish a communication between plan code and other code. There exist external calls to read and write configuration entries from outside a plan.

C. The Actors' Life Cycle

The following pseudo code sketches the main loop for executing a plan.

```
start all global supervisors in threads
curAct=first actor
while curAct≠terminate {
   start all supervisors of curAct in threads
   read input from configuration
   try {
      state=execute curAct
      write output of curAct to configuration
      curAct=actor connected by state
   }
   catch (interrupted by supervisor) {
      event=event of interrupting supervisor
      write output of supervisor to config.
      curAct=actor connected by event
   }
   terminate all non-global supervisor threads
}
terminate all global supervisor threads
```

The code carefully keeps care, always to have a single actor active. As a consequence, the code has to respect critical sections not indicated by the pseudo code above:

- The actor only writes to the configuration, if it terminates without interruption. Thus, after executing an actor, writing to configuration and detecting the next actor cannot be interrupted by a supervisor.
- Only one supervisor can interrupt an actor, if multiple supervisors create an event simultaneously, only one creates output and selects the next actor.
- Any writing to output is first applied to a shadow copy of the configuration. Not before the terminating component (actor or interrupting supervisor) is identified, the output is applied to the real configuration.

If an actor is a sub-plan, the plan execution is applied recursively. This means, from the view of the upper plan, the sub-plan actor is active and from the viewpoint of the sub-plan, a certain actor inside the plan. This affects, how supervisors signal an interruption, as now we have multiple main loops active. As a result, each supervisor has to know the active actor which can be interrupted. Also global supervisors in sub-plans have to ensure not to interrupt the upper plan. However, all required information to control this behavior can easily be retrieved from a plan.

To support plan execution, the underlying runtime system has to offer threads. Moreover, real-time conditions have to be respected, especially for supervisors. E.g. a typical supervisor could be: check every 1 ms, whether the tactile collision sensor detects an obstacle. Thus, the supervisor has to express the timing condition and the real-time system has to ensure that every supervisor thread is called in time.

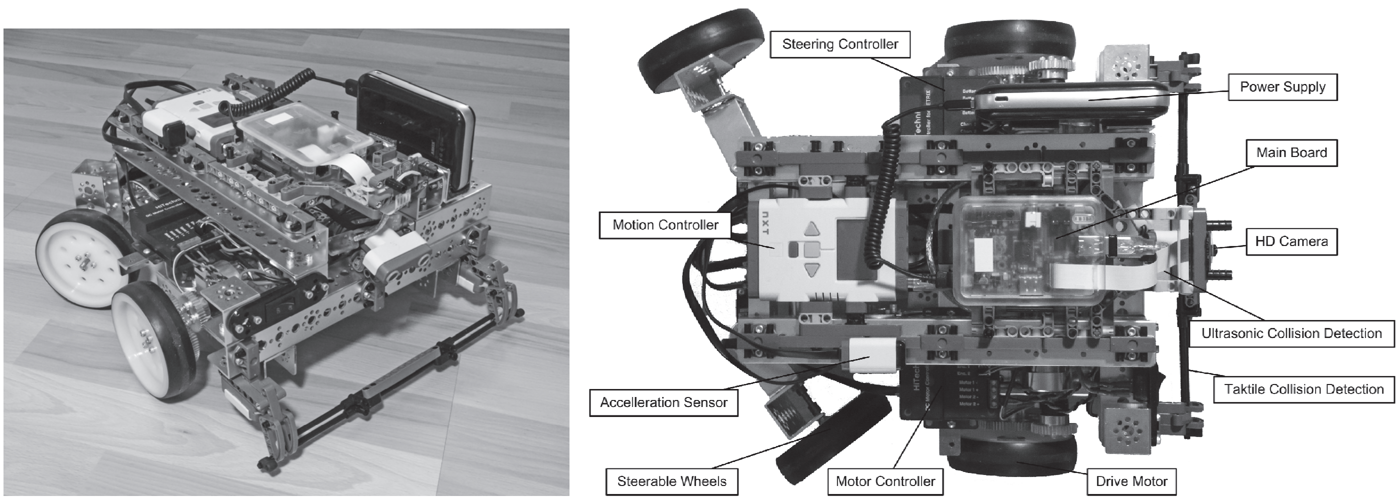

Fig. 4. The Carbot

IV. The Sample Implementation

We fully implemented a tool chain based on our approach. A desktop application contains the graphical editor, code generator and upload tools to the target system. Code generation currently produces Java code. We implemented Action Flow Plans to e.g., find an object and drive to its location, navigate around an obstacle, perform certain driving patterns or follow a wall (described above).

We executed the application on our Carbot platform (fig. 4). It has a size of 35 cm x 40 cm x 22 cm and a weight of 4,7 kg. It is able to run with a speed of 31 cm/s (1.1 km/h). A unique wheel configuration allows to independently steer two wheels. It is possible to drive curves like a car, but also turn in place.

Carbot detects collisions with a tactile senor and an ultrasonic distance sensor. To recognize the environment, Carbot's HD camera is able to capture images. Image processing is based on the *OpenCV* framework [9] that fully is executed onboard. We apply an Optical Flow approach to identify and follow objects in the image stream [2][6].

It is easy to attach special sensor configurations on top of the robot. E.g. a turnable distance sensor can 'look around' and construct a distance map. Further sensors can e.g. detect the direction of an infrared source or read optical properties (e.g. the color) of nearby objects.

Carbot operates an own WLAN access point for uploading software and system maintenance. The entire software runs fully autonomously, i.e. after uploading no further computer is required. The main computing hardware is a Raspberry Pi. An additional system executes driving commands and relieves the main board from time-critical tasks.

Table I shows the generated source code for the plan in fig. 2. The code mainly creates instances of the respective actors and supervisors and links them according to plan connections. The code makes use of an instance `ee` (execution environment) that provides access to all available sensors and actors.

V. Conclusions

In this paper we presented the Actor Flow Plans that introduce a new level of abstraction to highly event driven applications. The plan models a 'normal' execution by actors and all types of interruptions by supervisors. In addition, the flow of actions and the flow of data are modeled in a single representation.

As the plans are above source code level, the same plan can be used for different target systems. If two target platforms support the same actors and supervisors, we only have to implement the corresponding toolbox support. This makes it easy to share algorithms between different platforms.

In the future we want to enable live debugging in the plan representation. For this, an online communication to the target system has to be established (e.g. via WLAN) and the code generation has to produce additional code.

References

[1] Bagnall, B. 2013: Maximum Lego NXT, Variant Press

[2] Beauchemin, S. S, Barron, J. L, 1985: The Computation of Optical Flow, ACM Computing Surveys, Vol. 27, No. 3, September 1995, 433-467

[3] Bishop, R. H. 2014: LabView, Prentice Hall

[4] Brooks, R. A. 1986: A Robust Layered Control System for Mobile Robot, IEEE Journal of Robotics and Automation, RA-2, April 1986, 14-23

[5] Crow , A., Crow , G. 2013: Learning to Program with RobotC, Eagle Trail Press

[6] Gilad, A., 1985: Determining Three-Dimensional Motion and Structure from Optical Flow Generated by Several Moving Objects, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7, Issue 4, 384-401

[7] Griffin T. 2010: Art of Lego Mindstorms NXT-G Programming, No Starch Press

[8] Jones, J. L., 2004: Robot Programming – A Practical Guide to Behavior-Based Robotics, Mcgraw Hill

[9] Suarez, O. D., Carrobles, M. F., Enano, N. V., Garcia, G. B., Gracia, I. S. 2014: OpenCV Essentials, Packt Publishing

TABLE I. GENERATED CODE FOR WALLFOLLOW

```
// **** Create Actor instances ****
Forward searchFirstWall=new Forward(
     ee.getNavigator(),"100000");
Forward backupCollision=new Forward(
     ee.getNavigator(),"-walldist");
SearchWall findWallAngle=new SearchWall(
     ee.getUltrasonic(),ee.getRotator(),
     ee.getRotatingTelemetry(),"wallangle");
Turn parallelToWall=new Turn(
     ee.getNavigator(),"wallangle");
WallForward followWall=new WallForward(
     ee.getNavigator(),ee.getUltrasonic(),
     ee.getRotator(),ee.getRotatingTelemetry(),
     "walldist","walldist",
     "1","20","500");
Turn turnAfterWallLost=new Turn(
     ee.getNavigator(),"-60");

// **** Create Supervisor instances ****
CheckEscapeButton escape=new CheckEscapeButton();
CheckTouchSensor touch=new CheckTouchSensor(
      ee.getTouchSensor());
CheckDrivingLane checkLane=new CheckDrivingLane(
      ee.getUltrasonic(),ee.getRotator(),
      ee.getRotatingTelemetry(),
      "walldist","10","10","25");
CheckTouchSensor wallTouch=new CheckTouchSensor(
      ee.getTouchSensor());

// **** Create Plan ****
plan=new Plan(ee.getConfiguration());
plan.defineFirstActor(searchFirstWall);
plan.defineSupervision(escape,
     CheckEscapeButton.EVENT_ESCAPEBUTTONPRESSED,
     Plan.ACTION_TERMINATE);

// **** Define rules for Actor searchFirstWall ****
plan.defineSubsequentActor(searchFirstWall,Forward.
     STATE_FORWARDSUCCESSFULL,searchFirstWall);
plan.defineSupervision(searchFirstWall,touch,
     CheckTouchSensor.EVENT_TOUCHSENSORPRESSED,
     backupCollision);
```

```
plan.defineSupervision(searchFirstWall,checkLane,
     CheckDrivingLane.EVENT_LANENOTFREE,
     findWallAngle);

// **** Define rules for Actor backupCollision ****
plan.defineSubsequentActor(backupCollision,
     Forward.STATE_FORWARDSUCCESSFULL,
     findWallAngle);

// **** Define rules for Actor findWallAngle ****
plan.defineSubsequentActor(findWallAngle,
     SearchWall.STATE_WALLFOUND,parallelToWall);
plan.defineSubsequentActor(findWallAngle,
     SearchWall.STATE_NOWALLFOUND,
     turnAfterWallLost);

// **** Define rules for Actor parallelToWall ****
plan.defineSubsequentActor(parallelToWall,
     Turn.STATE_TURNSUCCESSFULL,followWall);

// **** Define rules for Actor followWall ****
plan.defineSubsequentActor(followWall,

WallForward.STATE_FORWARDSUCCESSFULL,followWall);
plan.defineSubsequentActor(followWall,
     WallForward.STATE_LANENOTFREE,findWallAngle);
plan.defineSubsequentActor(followWall,
     WallForward.STATE_WALLLOST,turnAfterWallLost);
plan.defineSupervision(followWall,wallTouch,

CheckTouchSensor.EVENT_TOUCHSENSORPRESSED,backupCol
lision);

//  ****  Define  rules  for  Actor  turnAfterWallLost
****
plan.defineSubsequentActor(turnAfterWallLost,
     Turn.STATE_TURNSUCCESSFULL,searchFirstWall);

...

plan.execute();   // Execute the plan
```