Proc. of WESIC 2001, University of Twente, The Netherlands, 27-29 June 2(

BEHAVIOUR-BASED CONTROL FRAMEWORK FOR AN AUTONOMOUS MOBILE ROBOT

Albert L. Schoute

Dept. of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands
email: a.l.schoute@cs.utwente.nl

Abstract

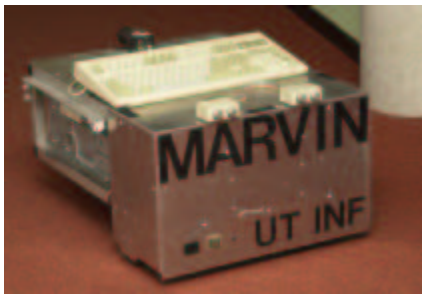
This paper presents the concept of an object-oriented software framework that provides a scheduling and execution environment for behaviour-based control of an autonomously navigating mobile robot. The framework is build around a basic class for 'behaviours' and for associated 'situations' that handle exceptional conditions. New control methods and navigation strategies are easily incorporated and tested by extending this framework.

1 Introduction

Planning and control functions within intelligent, embedded systems are generally divided over multiple software layers. The lower layer(s) of the software hierarchy typically contain(s) the basic functions that directly control the hardware, i.e. the sensors and actuators. The top layer(s) handle(s) the long term planning and usually a high-level command interface. In between one can identify one or more intermediate layers that are responsible for fulfilling the current goals or tasks.

In this paper we report about a general concept for the task scheduling framework at this intermediate level. In the context of an autonomous mobile robot the tasks to do are sequences of behaviours the robot has to perform in order to reach its goal. Behaviours are primitive actions such as moving through a hallway, entering a doorway, following a moving object, approaching some landmark, or just moving to some position. Behaviours are scheduled and executed like tasks in an operating system. In principle they run to completion, but – in reaction to circumstantial conditions - they may be pre-empted or suspended. Exceptional situations may be associated with behaviours to catch unexpected conditions in parallel to the normal behaviour execution. A situation-monitoring process is introduced as detection and exception handling mechanism.

The framework that we describe is part of a control system for an experimental robot vehicle, named Marvin [Koetsier 1997]. Marvin is a low-budget, PC-based vehicle that contains all essential elements to serve as a test-bed for autonomous system development. The vehicle can drive physically unconnected by means of its own battery-power-supply, two independently driven wheels, ultrasonic distance sensors and a high-resolution CCD-



camera with PCI-bus frame-grabber. A wireless LAN-connection enables remote monitoring and control. The control software runs on the Linux operating system. It has an object-oriented structure (written in Gnu-C++) and uses multithreading.

The objective of the test-bed is to explore techniques by which robot vehicles can navigate in office environments using natural properties of the building. Although we allow the control program to exploit pre-knowledge of static building features, behaviours have to cope with the actual situation in a robust, reactive manner. The behaviour-based control framework creates an ideal context for experimentation: it is easy to program and test new behaviours. Object-orientation cares for the encapsulation of the basic properties of the behaviours and associated exceptional situations that must be traced. The control framework automatically provides the context in which these behaviours and situations are scheduled and executed. For example, position information in absolute or behaviour-relative coordinate systems is maintained and always accessible. Also remote monitoring and interaction is supported in a general way.

2 The software control system

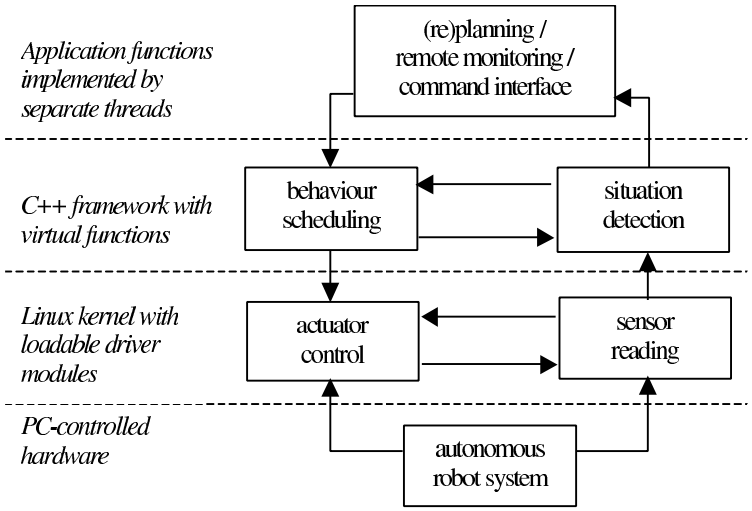


Figure 1 Schematic overview of the software control system

The global structure of Marvin's control software is shown in Figure 1. Drivers for the ultrasonic sensors, the servo-motors and the frame-grabber are written in C as loadable kernel modules. These drivers implement a file-oriented device interface that can be accessed by standard system calls (like open, read, write, control, close). The Linux kernel supports multithreading by creating Posix-compliant threads (*pthreads*) as kernel processes [Beck 1997]. Separate threads have been introduced for behaviour execution, situation monitoring, user interaction, planning, remote connection handling and image capturing.

In view of the complex way in which threads interact, the two top layers heavily rely on the object-oriented approach. Object classes for behaviours and situations play a central role in the control framework. The actual control of the robot depends on the currently active 'behaviour' and 'situation' objects. Object orientation facilitates a unified treatment of these objects by means of a common base class, whereas any specific control is detailed in a particular "derived" class. Common but dissimilar functions are

implemented by virtual functions. The control program contains many shared components accesses by multiple threads. Classes help to structure these components by aggregating common data and functions and providing clearly defined interfaces. Shared interfaces in the control software are, for example, data stores containing recent sensor values, the robot status, the list of scheduled behaviours and the user control panel. The pthread-package supplies synchronisation functions for exclusive access to class objects according to the monitor concept [Silberschatz 2000].

3 Behaviour-based control

The base class BEHAVIOUR defines the basic, common properties of behaviours. All implemented behaviours have a "control function" in common. The function *control* of the current behaviour will at any instance determine the momentary behaviour of the robot. Behaviours are instantiated dynamically, which could happen in all sorts of circumstances dependent upon the application. The instantiation may be part of a pre-planned series of actions, but even likely be the result of remote intervention, sensor-based reactions or other unforeseen situations. New instances of behaviours can be entered for scheduling by the current behaviour or any situation handler. A switch of behaviour may occur either by termination or by interruption.

3.1 Behaviour scheduling

The execution of behaviours, queued on a global *behaviour list*, is delegated to a separate *execution-handler* thread. New behaviours, for example entered by the planner, are typically placed at the tail of the list. A behaviour will in general run to completion before the next behaviour is activated. The occurrence of special situations could, however, disturb this normal FIFO-order. Behaviours can pass through a number of states as shown in Figure 2. The scheduling is organized such that the head of the behaviour list is always taken as the next behaviour. The state of a behaviour on the list could be either INITIAL (not been in action before) or RESUME (interrupted but still to be completed). A behaviour that is currently active has state EXECUTE. It can be set in state DESTROY for definite removal or in state SUSPEND for temporary pre-emption. In the latter case other behaviours may have been instantiated and inserted in the list before the pre-empted behaviour. In this way the scheduling works as exception mechanism: other behaviours can cope with the situation before the interrupted behaviour resumes. It allows for stack-wise (LIFO) scheduling in exceptional circumstances. In practice, this works in a natural and effective way. A dangerous situation may lead to a temporary interruption by the IDLE behaviour such that the robot does not move as long as the situation remains. Or, any behaviour could be overruled by the REMOTE_CONTROL behaviour temporarily. The IDLE behaviour is also added and executed in case of an empty behaviour list. For example at system start-up this is the first behaviour that becomes active.

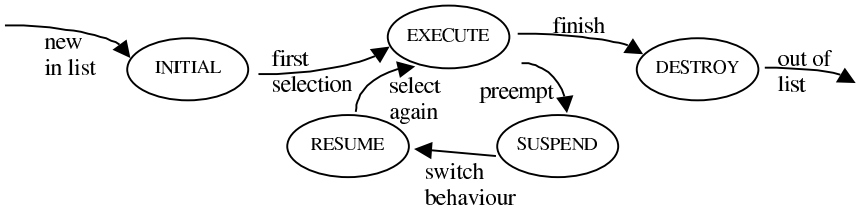


Figure 2 State transitions of behaviours in the behaviour list

The execution-handler thread performs the execution of the current behaviour in a cyclic, periodically timed loop. It calls for the function *control* of the current behaviour and pauses according to some fixed time interval. The function *control* is declared in the base class BEHAVIOUR as a virtual function, which means that the call is bind at execution time to the specific implementation of this function within the particular "derived" behaviour class. An important aspect of the general functioning of behaviours concerns motion control and position tracing, which is treated in a next section.

3.2 Situation handling

Besides behaviour execution, a general mechanism is added for the detection of exceptional situations not covered by the behaviours them selves. An independent *situation-handler* thread runs concurrently with the execution thread. The separate treatment of unpredictable circumstances with respect to the normal, expected behaviour, highly contributes to the flexible and robust operation of the autonomous system. Behaviours are freed from anticipating, at any instance, all kinds of special situations. The same situations may occur during many behaviours and are therefore handled at best in a separate and independent fashion.

Similar to the control function of behaviours, for situation handling a virtual function *do_situation* is declared in a base class SITUATION. If a situation instance is active the function is called periodically to detect some special condition and react on it. Any behaviour has a list of associated situations. These objects of derived situation classes are typically created at behaviour instantiation. It is the task of the application environment (for example the planner) to associate a behaviour with the appropriate situation objects. For safety reasons, certain important situations are always associated with behaviours, like the SAFETY_CHECK (checking collision) and the REMOTE_CONTROL situation (checking remote intervention). In case of a transient risky situation (some person is passing by) the current behaviour is pre-empted by the IDLE behaviour, and is later on resumed automatically. This behaviour-switching appears as a natural reaction of the robot. Optional situations are for instance the OUTSIDE_RANGE situation (to cancel a behaviour that carries the robot outside some radius) or LANDMARK situation (to detect some environment feature).

The situation handling thread scans the situation list of the current behaviour and calls for the function *do_situation*. Situation scanning can be used also for independent monitoring or tracing of certain conditions and variables. The situation function may contain output statements that display state information on the control panel or log data for later examination.

4 Motion control and position tracing

Motion is of course a dominant factor in case of mobile robot control. Most of the behaviours will be related to motion manoeuvres and sensor-based navigation strategies. In fact a great advantage of the framework is that it offers an environment for experimentation in which alternative motion behaviours easily can be tested and compared. Common aspects of motion behaviours like speed control and position localization are supported already by the system and made available in the base class. Any specific behaviour inherits the basic motion control properties; only the particular elements have to be added within a derived class declaration. Motion control and position information are handled by a hierarchy of layers as depicted in Figure 3.

The kernel module for the motor device does the direct i/o to actuate the wheel motors and to read the tachometers that measure the rotation speed of the wheels. The wheel configuration of Marvin constitutes a differential drive mechanism [Dudek 2000]. The two independently driven motors of the back wheels are regulated by feedback control to satisfy the linear and angular speeds as required by the upper layers. The control loop is activated every 10 milliseconds by a system timer. Odometry is used to estimate the robot's position. According to the kinematics, relative displacements are calculated from the observed rotation speeds of the wheels.

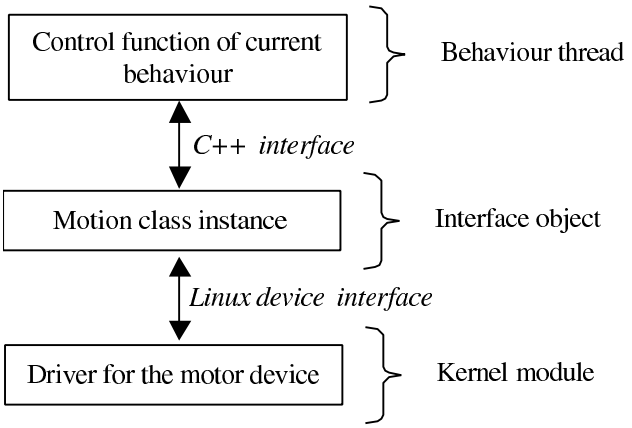


Figure 3 Software components involved with motion control and position tracing

The "motion class" object contains motion state variables (position, speeds acceleration) and functions to access the motor device by *read*, *write* and *ioctl* system calls. Calling a function *refresh_motion_state* keeps the state variables up to date: it performs a read system call to obtain the most recent values from the motor device driver. By means of a function *set_speed* the desired linear and angular speeds are written to the motor device driver as setpoint references. Furthermore the global vehicle position maintained by the motor driver may be reset or corrected by functions *set_position* or *set_diff_position*. Correction of the global position state is necessary because accumulation of errors makes the absolute position estimation inaccurate over longer distances. It is the aim of experiments with sensor-based navigation to observe natural building features and use these for absolute position localization. The robot's pose is represented by a coordinate frame (a class object of type FRAME). The pose consists of the x, y position and the orientation (i.e. the heading of the robot). In fact coordinate frames are relative notions: the robot's pose is given by the actual placement of its "body frame" relative to some reference frame in the ground plane. The class FRAME provides functions for frame transformations like rotate and translate.

Motion behaviours generally concern relative manoeuvres with respect to some starting position. The base class BEHAVIOUR contains a *local frame centre* that is defined at the behaviour's instantiation. During behaviour execution the current pose relative to this local frame is maintained by calling a function *refresh_local_pose* before the control function of the current behaviour is executed. The current pose is calculated by applying an inverse frame transform on the robot's pose with respect to the local centre. In this way motion behaviours can be programmed easily in a uniform manner without bothering about the actual positioning at the moment of application. The only concern of the behaviour's control function is to adapt the linear and angular speed parameters according to the motion state in the local frame and, probably, according to available

sensor data or interactively defined global variables. Some simple examples of particular control function are shown in Figure 4.

```
void move_bhv::control(float *lin_speed, float *ang_speed)
{ /* follow reference motion if enough free space around */
  *lin_speed = *ang_speed = 0; /* default: don't move */
  if ((lin_ref_speed > 0 && sensors->free_range(FRONT) > 0.2)
      || (lin_ref_speed < 0 && sensors->free_range(BACK) > 0.2))
     *lin_speed = lin_ref_speed;
  if (sensors->free_range(LEFT) > 0.2
      && sensors->free_range(RIGHT) > 0.2)
     *ang_speed = ang_ref_speed;
}

void remote_control_bhv::control(float *lin_speed, float *ang_speed)
{
  if (!marvin_remote) { /* check termination */
     state = STATE_DESTROY;
     return; }
  *lin_speed = remote_lin_speed;
  *ang_speed = remote_ang_speed;
}

void turn_bhv::control(float *lin_speed, float *ang_speed)
{ /* turn until current orientation becomes zero */
  float abs_diff = fabs(curr_pose.phi);
  bool  sign = (curr_pose.phi > 0.0);
  if (abs_diff< 0.02) { /* (almost) reached */
     state = STATE_DESTROY;
     return; }
  if (abs_diff) > 1.0)
     *ang_speed = (sign ? -0.6 : 0.6);
  else  if (abs_diff > 0.5)
           *ang_speed = curr_pose.phi * -0.6;
        else  *ang_speed = (sign ? -0.3 : 0.3);
  *lin_speed = lin_ref_speed;
}
```

Figure 4 Control functions of some derived behaviour classes

5 Remote user interaction

During operation of the robot control program actual state information is regularly written to the standard terminal output. By means of cursor control these output is presented at fixed places on the screen according to a pre-defined control panel layout (see Figure 5). Items displayed are amongst others the current behaviour, its centre frame, the actual robot position and speeds, active situations and, optionally, a trace of the motor control variables, ultrasonic sensor readings or a primitive map of the traversed path. User interaction is possible due to a *user-input-handler* thread that reads the terminal input. By means of a simple tree-based menu selection mechanism the user is able to start and stop single behaviours, inspect the actual behaviour list and do many other things (putting motors on/off, resetting the global position, starting image acquisition, etc.). It can also invoke existing script files that contain series of behaviours with associated situations. Such script files are interpreted by a planner thread. The planner will instantiate and initialise the corresponding behaviour and situation objects.

Normally, the Marvin robot drives around without a keyboard or monitor connected to the onboard computer. The only channel of communication with the Linux operating system is a wireless TCP/IP connection provided by a Wavelan-card (Lucent). The

connection uses radio transmission with communication speeds up to 2 Mb/s over an indoor range of maximal 100 meter. By establishing a Telnet-session, the robot control program can be started and controlled remotely.

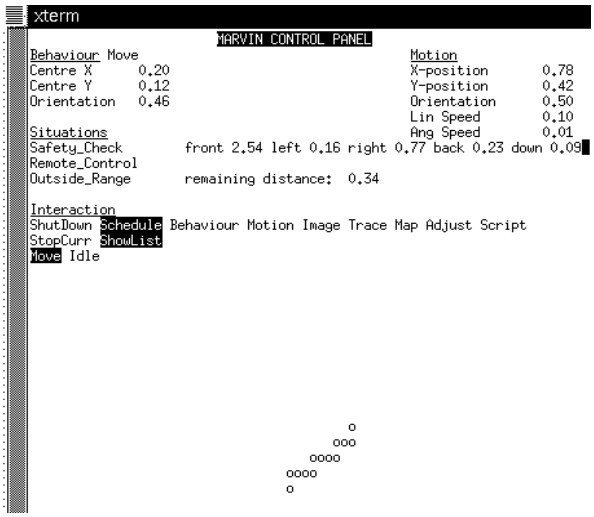


Figure 5 Remote session with Marvin's control program

A more advanced, graphical user interface for monitoring and control, shown in Figure 6, has been developed in connection to vision-based navigation experiments. A client-program (for Windows) interacts via a TCP/IP socket connection (also using the wireless network) to a server thread in Marvin's control system. It can display camera-images (at a rate of 4 images/sec) and other sensor data. It can also start and stop behaviours, for example a recognition behaviour to detect and follow a moving pattern. It may even overrule the vehicle motion and steer Marvin remotely by activating the REMOTE_CONTROL behaviour.

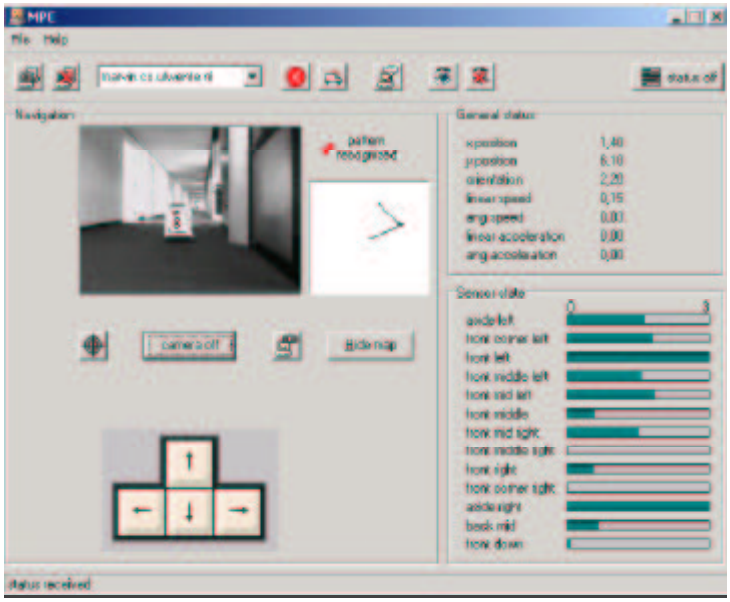


Figure 6 Graphical interface of remote monitor program

6 Experiments

Experiments have been carried out with respect to motion control strategies for sensor-based navigation. Besides the ultrasonic distance sensors, computer vision is used to enhance the ability of position sensing. Additional threads and classes have been introduced to maintain a real-time "store" of camera images that can be accessed (read-only) at any time by one or more behaviours or threads (like the server thread fulfilling remote image requests). Images in the store are claimed during processing and have to be freed explicitly to allow buffer reuse.

Examples of vision-related behaviours are the "moving pattern tracking" behaviour and a "free space search" behaviour. The latter one detects the visual edges of the ground floor and drives according to a free-space map that is derived from it (see Figure 7). The behaviour of "driving through a hallway" has been explored by different sensor approaches. An observer-based controller for position tracking has been employed successfully using ultrasonic wall-distance measurements only [Siers 2000]. Position estimation based on Kalman filtering of multiple sensor data has been investigated by [Klein 2000]. It exploits the fact that the position of the "vanishing point" of the hallway in the camera image reveals the heading direction of the robot [Zoghbi 2000].

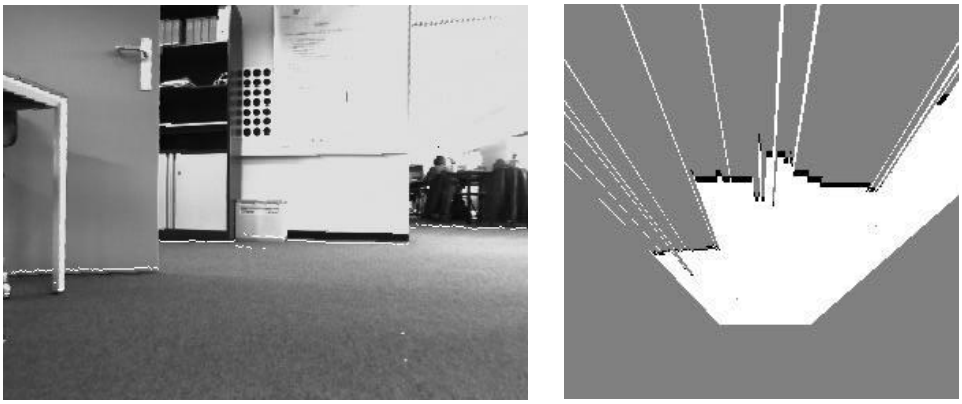


Figure 7 Recognition of the floor edges (as shown by the white pixels) with its corresponding free-space map

7 Conclusion

The presence of the general framework has alleviated the required effort for experimentation considerably. Both the multithreading facility of Linux and the object orientation keep the development of the robot control program manageable. New components can be implemented independently, provided that they conform to the existing interfaces. To introduce a new behaviour the control program needs to be extended only at clearly isolated places. Mainly a new (derived) behaviour class has to be programmed with its own control function. New situation handlers are fitted easily in the existing scheme. Components of the control system, built earlier, are in general easily reusable and/or adaptable for new experiments and application contexts. Behaviours with associated situations can be simply tested interactively by using the existing control panel interface operated via a remote telnet session. During operation, state information is permanently logged. Extra logging of variables for testing purposes can be added to the behaviour's control function or to situation handlers.

The capabilities of the autonomous robot have been extended over the years without a need for a major revision of the basic framework. The behaviour-based control mechanism has shown to be very versatile in combination with the client-server approach for remote monitoring and control, introduced at a later stage.

The use of shared resources (like the image store) by multiple threads requires careful synchronisation. However, once the appropriate access functions to shared class objects are written correctly (by using exclusion and condition synchronization as provided by the monitor concept) the pitfalls of concurrency are hidden and do not burden the application context.

References

Beck, M. et al (1997), *Linux Kernel Internals*, 2nd Ed., Addison Wesley, ISBN 0-201-33143-8

Dudek, G. and M. Jenkin (2000), *Computational Principles of Mobile Robotics*, Cambridge University Press, ISBN 0-521-56876-5.

Klein, A.J. (2000), *Position determination of an autonomous robot vehicle based on multiple sensor information*, Master's thesis, DIES-2000-03, Dept. of Computer Science, Univ. of Twente.

Koetsier, G.H. (1997), *Supervisory control framework for an autonomous service vehicle*, Master's thesis, SPA-97-019, Dept. of Computer Science, Univ. of Twente.

Siers, M. (2000), *Controller design for tracking and stabilization of a two-wheeled mobile robot*, Master's thesis, Dept. of Applied Mathematics, Univ. of Twente.

Silberschatz, A. et al (2000), *Applied Operating System Concepts*, John Wiley & Sons, ISBN 0-471-36508-4.

Zoghbi, P. (2000), *Image processing for self-localization of an autonomous vehicle*, IAESTE Practical Training Report, Dept. of Computer Science, Univ. of Twente.