Torsten Evers

Ausgang gesucht!

Komplexe Verhalten für den c't-Bot selbst entwickelt

Bisher dreht sich der c't-Bot, spurtet vorwärts oder fährt Slalom. Nun soll er lernen, einen Ausweg aus einem komplexen Labyrinth zu finden. Dazu braucht es einen geeigneten Algorithmus und ein wenig Wissen über den Aufbau des c't-Bot-Frameworks.

Zum Durchqueren eines Labyrinths existieren diverse Algorithmen, die ihre Aufgabe unterschiedlich schnell und zuverlässig lösen. Einfache Vertreter sind der „Höhlenforscher-" oder auch der Pledge-Algorithmus. Beide machen sich eine Eigenschaft bestimmter Labyrinthe zu Nutze: Liegen Ein- und Ausgang in der Außenwand, erreicht man immer das Ziel, wenn man kontinuierlich an einer Wand entlangläuft. Der Höhlenforscher-Algorithmus, dessen Umsetzung für den c't-Bot dieser Artikel Schritt für Schritt als Beispiel vorstellt, verfolgt diese Strategie. Sie kann jedoch in Labyrinthen scheitern, die vollständig umrundbare Elemente wie Pfeiler enthalten oder in denen der Startpunkt nicht an einer Außenwand liegt.

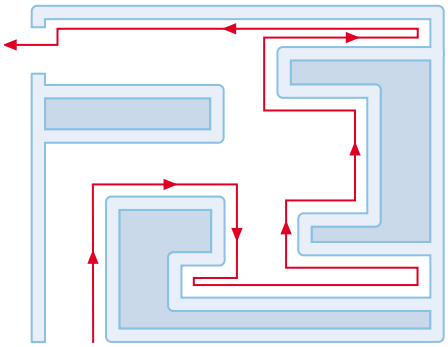
Gerät der Bot an ein solches Hindernis, folgt er der Wand und hängt somit in einer Endlosschleife fest. Um dies auszuschließen, geht der Pledge-Algorithmus einen Schritt weiter und implementiert einen Winkelzähler. Diesen erhöht er bei jeder Drehung um den entsprechenden Winkel. Erreicht der Bot wieder die Ausgangsfahrtrichtung und hat dabei eine 360°-Drehung vollzogen, löst er sich von der Wand. Für die hier betrachteten, Labyrinthe reicht die Höhlenforscher-Variante aus.

Roter Faden gesucht

Will man komplexe Aufgaben wie den Höhlenforscher auf dem c't-Bot implementieren, untergliedert man diese zunächst in Teilschritte. Danach zeigt sich, ob in der Code-Basis die nötigen Routinen bereits vorhanden sind oder ob man sie neu programmieren muss. Dabei kann man auf vieles zurückgreifen, was im Rahmen der Artikelserie bereits vorgestellt wurde. Der Kasten „Wie sag ich es meinem Bot?" erklärt noch mal ausführlich den Umgang mit dem c't-Bot-Framework.

In Labyrinthen tauchen folgende Grundelemente immer wieder auf: gerades Wandstück, Ecken und Kanten (siehe Grafiken auf Seite 237). Um bei jeder dieser Strukturen den richtigen Weg zu finden, muss der Bot jeweils mehrere Aktionen ausführen. Der Aufbau des Roboters bestimmt maßgeblich, welche Bewegungen er dazu durchführen muss. Da der c't-Bot keine seitlichen, sondern nur frontale Abstandssensoren besitzt, muss er sich immer wieder zur Wand umdrehen, um diese zu sehen. Zudem ist weder mit dem realen Bot noch im c't-Sim absolute mathematische Genauigkeit bei Drehungen oder Fahrstrecken erreichbar. Mit Ungenauigkeiten muss der Algorithmus also klarkommen.

Bevor der Bot jedoch mit der eigentlichen Arbeit loslegen kann, muss er zuerst die Wand finden, der er folgen soll. Die hier vorgestellte Lösung geht davon aus, dass der Bot – wie im Bild zu sehen – neben einer Wand mit Blickrichtung zum Labyrinth startet. Dabei spielt es keine Rolle, auf welcher Seite die Wand steht. Die Abstandssensoren erkennen die Wand problemlos, sofern sie sich in ihrem Messbereich (10 bis 80 cm) befindet. Ist die Entfernung zur Wand zu gering, schätzt der Bot diese falsch ein. Dies resultiert aus der Kennlinie der Abstandssensoren. Kommt der Bot einem Hindernis näher als der Untergrenze des Messbereichs der Sensoren, steigen die gemessenen Werte wieder, anstatt entsprechend der abnehmenden Entfernung zu sinken [3]. Vorausschauendes Fahren empfiehlt sich daher nicht nur für Menschen.



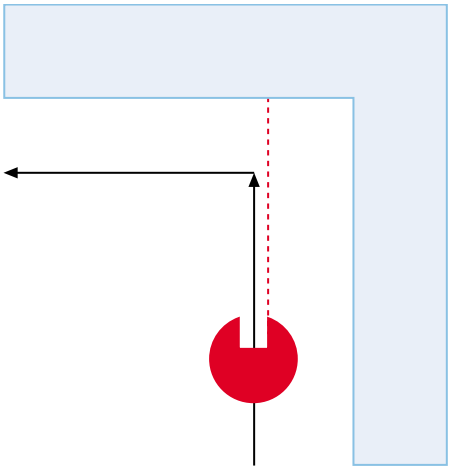
Folgt der Bot in einem Labyrinth kontinuierlich einer Außenwand, so findet er immer zum Ausgang.

Der Algorithmus dreht den Roboter zuerst nach rechts und sucht dort nach einer Wand. Findet er keine, wiederholt er das Prozedere auf der gegenüberliegenden Seite. Ist auch hier keine Wand, fährt er eine kurze Strecke vorwärts und wiederholt die Suche. Einmal gefunden, soll der Bot der Wand folgen. Dazu fährt er eine Strecke, die seinem Durchmesser entspricht, voran und dreht sich dann in Richtung Wand. Eine 45°-Drehung eignet sich gut, um auch Kanten früh zu erkennen. Die benötigte Funktion gibt es bereits im Framework:

```
bot_turn(data,45);
```

Den Parameter `data` benötigt das c't-Bot-Framework intern (siehe Kasten). Ungenauigkeiten bei der Drehung gleicht der Bot aus, indem er die Entfernung zur Wand korrigiert und sich anschließend zurück in Fahrtrichtung dreht.

Das zweite Element des Labyrinths, die Ecke, findet der Bot von selbst, da seine Distanzsensoren die nahende Wand früh erkennen, während er auf sie zufährt. Aus der gemessenen Entfernung ergibt sich nach Abzug der optimalen Distanz des Roboters



Mit seinen Abstandssensoren erkennt der c't-Bot eine Ecke bereits frühzeitig.

zur Wand die Fahrstrecke, die er noch ohne weitere Drehungen zurücklegen kann.

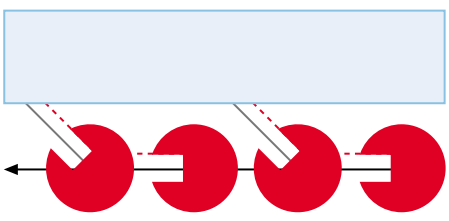
```
bot_drive_distance(data,0,BOT_SPEED_NORMAL,
                   (distance-OPTIMAL_DISTANCE)/10);
```

Neben dem schon erwähnten Parameter data erwartet bot_drive_distance eine Krümmung (hier 0), die Geschwindigkeit und eine Strecke in Zentimetern. Da die Sensoren Entfernungen in Millimeter liefern, teilt man den errechneten Wert noch durch 10. Die eigentliche Arbeit übernimmt dann eine Routine aus dem Verhaltens-Framework (siehe Kasten). Die Tabelle zeigt alle bereits im Framework enthaltenen Verhalten mit ihren Botenfunktionen.

Wenn der c't-Bot bei einer Prüfdrehung keine Wand mehr findet, steht er an einer Kante. Da die genaue Fahrstrecke bis zu ihrem Erreichen stark variieren kann, muss das Verhalten sie berechnen. Die Grundlage für diese Berechnung bildet der Winkel, um den der Bot sich drehen muss, bis die Wand wieder in seinem Sensorbereich erscheint.

In medias res

Die Überprüfung, ob der Bot noch neben der Wand fährt und sein Abstand dazu stimmt, findet regelmäßig statt. Daher bietet sich die Implementierung in einem Hilfsverhalten an. Es kontrolliert dabei nicht nur, ob die Wand noch in Sichtweite ist, sondern führt auch Korrekturen für Entfernung und



Fehlende seitliche Sensoren macht der Bot durch regelmäßige Drehungen zur Seite wett.

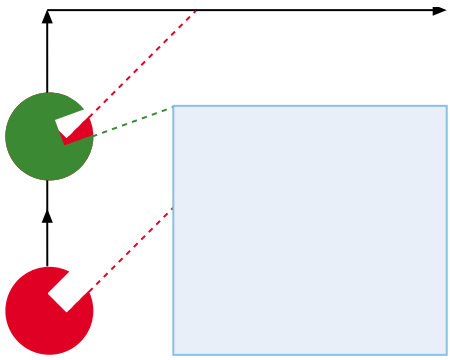
Winkel zur Wand durch. Dazu fährt der Bot je nach Entfernung von der Wand vor oder zurück, nachdem er sich um 45° gedreht und die Entfernung gemessen hat. Vor der Drehung in die Ausgangsposition ermittelt eine Faustformel – die exakte Bestimmung ist zu aufwendig und aufgrund der begrenzten Genauigkeit der Fahrmanöver nicht sinnvoll – den Winkel. Oft eignen sich solche Lösungen sehr gut und bedeuten weniger Ressourcenverbrauch bei gleich guten Ergebnissen im Vergleich zu einer exakt berechneten Lösung. Der Winkelfehler des Bots zwischen zwei Drehungen lässt sich mit einem ganz einfachen Trick korrigieren: Nachdem der Bot in seiner Kontrollmessung die 45°-Stellung erreicht hat, misst er die Entfernung und merkt sie sich. Dann fährt er vor oder zurück, bis er die Wand in OPTIMAL_DISTANCE sieht. Bei der nächsten Drehung wiederholt er die Messung. Die Differenz der beiden Entfernungen hilft, den Winkelfehler zu korrigieren. Hat der Bot wirklich in beiden Fällen einen 45°-Winkel eingenommen, muss die Differenz null betragen. Eine positive Differenz bedeutet, dass der Bot sich bei der letzten Kontrollmessung nicht weit genug zurückgedreht hat. Für die Drehung zurück wird daher ein Korrekturwinkel ermittelt (2° pro 10 mm liefert gute Resultate).

```
if (lastDistance!=0) {
    turnAngle=turnAngle+
              (lastDistance-wall_distance)/5;
  }
```

Die Botenfunktion bot_check_wall() bewältigt diese Aufgaben und nimmt als Parameter die Richtung entgegen, in der der Bot die Wand vermutet. Zu guter Letzt fehlt noch eine Möglichkeit, wie beschrieben die korrekte Fahrstrecke beim Erreichen einer Kante zu ermitteln. Auch das übernimmt ein Hilfsverhalten, sobald bot_check_wall_behaviour() signalisiert, keine Wand in der gewünschten Entfernung zu sehen. Der Bot befindet sich zu diesem Zeitpunkt kurz vor oder bereits kurz hinter der Kante und steht parallel zur bisher verfolgten Wand. Das Hilfsverhalten muss nun den Bot so lange in die Richtung drehen, in der bisher die Wand zu sehen war, bis der Abstandssensor auf dieser Seite sie wieder wahrnimmt. Aus den für diese Drehung gefahrenen Radencoder-Schritten berechnet die Routine den Winkel:

```
measured_angle=(int16)((uint16)turnedSteps*360/
                       ANGLE_CONSTANT);
```

Dabei darf man den Wertebereich der in der c't-Bot-Software verwendeten Datentypen (int8, int16, uint8 und uint16) nicht vergessen. Der Type-Cast auf uint16 verhindert einen Überlauf für den Fall, dass der Bot die Wand nicht entdeckt und sich bis zum maximalen Winkel von 360° weiterdreht. Da eine komplette Umdrehung 102 Encoderschritte erfordert, ergibt die Formel hier einen Wert von 36 720 statt der für int16 erlaubten 32 767. Den Datentyp int sollte man gar nicht nutzen, da er auf PC und Mikrocontroller unterschiedliche



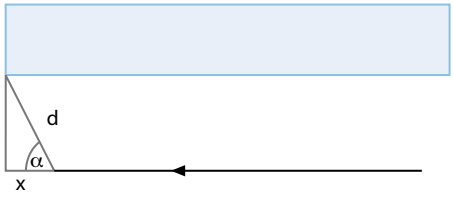
Da der Bot Kanten nicht ohne weiteres sehen kann, muss er regelmäßig anhalten, den Winkel zur Wand messen und dann die zu fahrende Strecke neu berechnen.

Wertebereiche hat. Er führt daher leicht zu Überlauffehlern.

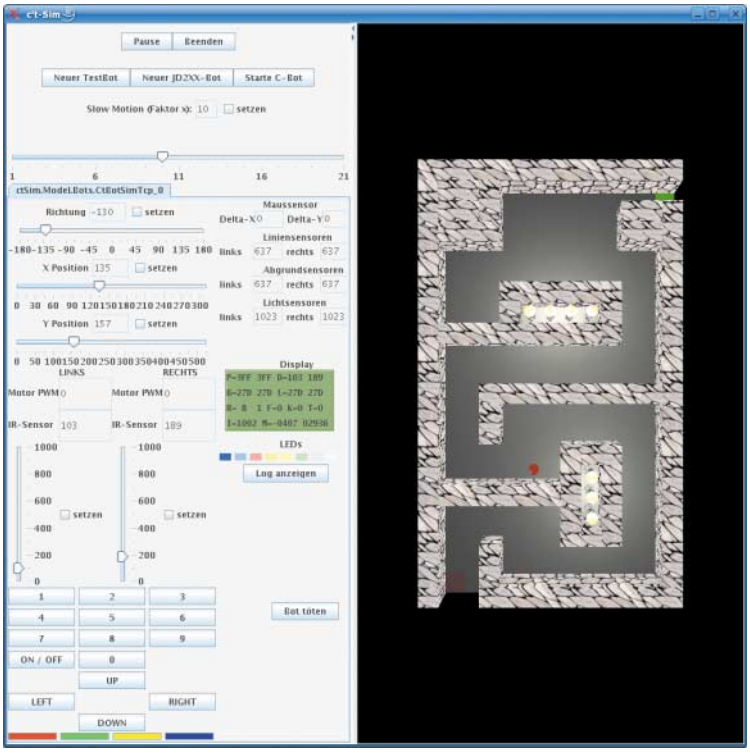
Das Hauptverhalten bot_solve_maze_behaviour() ermittelt zunächst die Wandposition und beginnt dann in einer aus drei Zuständen bestehenden Pseudo-Schleife der Wand zu folgen:

```
case SOLVE_MAZE_LOOP:
  mazeState=SOLVE_TURN_WALL;
  bot_drive_distance(data,0,BOT_SPEED_NORMAL,
                     BOT_DIAMETER);
  break;

  case SOLVE_TURN_WALL:
    ...
    mazeState=CHECK_CONDITION;
    bot_check_wall(data,followWall);
  break;

  case CHECK_CONDITION:
    if (wall_detected==True) {
      mazeState=SOLVE_MAZE_LOOP;
      break;
    }
    ...
    break;
```

Dabei verwendet es zur Ermittlung der Wandposition wie auch für die Prüfdrehungen bot_check_wall_behaviour(). Trifft der Bot auf eine Ecke oder Kante, wechselt das Hauptverhalten den Zustand. Für Kanten ermittelt bot_measure_angle_behaviour() den Winkel, aus dem sich die zu fahrende Strecke ermitteln lässt. Da diese Berechnung trigonometrische Funktio-



Nähert sich der Bot – auf einem geraden Kurs mit festem Abstand d zur Wand – einer Kante, wird der Winkel α kontinuierlich größer. Die noch zu fahrende Strecke, bis die Kante querab steht, beträgt x = d · cos α.



Der c't-Sim bietet eine ideale Testplattform für eigene Verhalten. So durchquert der Bot beispielsweise ein Labyrinth ohne Beeinflussung durch externe Störungen.

nen verwendet, benötigt sie die Mathematik-Bibliothek. Zur Schonung von Ressourcen sollte man Fließkommafunktionen und -variablen sparsam verwenden – der Controller muss diese immer durch viele einzelne 8-Bit-Integer-Operationen substituieren. Da alle Messdaten der Bots verrauscht sind, erzielt eine einfache (Integer-)Näherung oft ohnehin bessere Resultate als eine Rechnung mit vorgespielter Genauigkeit.

Umwelteinflüsse

Da das Verhalten selbst den Abstand zur Wand regelmäßig überprüft, versetzt es das Standardverhalten bot_avoid_col_behaviour() mittels der Funktion deactivateBehaviour() in einen Schlafzustand, um Beeinflussungen zu vermeiden.

Folgende Einträge in der Verhaltensliste sind nötig, damit der Bot durchs Labyrinth kommt:

```
insert_behaviour_to_list(&behaviour,new_behaviour
                (150,bot_solve_maze_behaviour,INACTIVE));
insert_behaviour_to_list(&behaviour,new_behaviour
                (43,bot_measure_angle_behaviour,INACTIVE));
insert_behaviour_to_list(&behaviour,new_behaviour
                (42,bot_check_wall_behaviour,INACTIVE));
```

Eine Taste auf der Fernbedienung soll das bot_solve_maze_behaviour() starten. Die Taste RC5_CODE_SELECT der Fernbedienung wechselt zwischen verschiedenen Verhalten aus der Verhaltensliste. Alternativ dazu kann man in der Datei rc5.c in der Funktion rc5_number(RemCtrlFuncPar *par) eine beliebige Zifferntaste mit dem gewünschten Verhalten belegen. Dazu trägt man die Botenfunktion bot_solve_maze() hinter derjenigen case-Anweisung ein, die zur ausgewählten Taste passt.

Ich will hier raus!

Es ist an der Zeit, das Programm im Simulator zu testen. Nach dem Start von c't-Sim und c't-Bot steht der virtuelle Gefährte bereits auf seinem Startfeld. Ein Druck auf die virtuelle Fernbedienungstaste aktiviert das Labyrinthverhalten. Vorsichtig tastet sich der Bot nun durch das ihm unbekannte Gefilde und folgt brav der Wand, die seinem Startpunkt am nächsten ist.

Was aber tun, wenn etwas schiefgeht? Oftmals ist es dann schon hilfreich, wenn man mehr als die puren Sensorwerte sehen kann, etwa Informationen über die berechneten Drehwinkel oder den Zustand, in dem der Bot feststeckt. Solche Daten lassen sich auf dem Display des simulierten Bots ausgeben. Die c't-Bot-Software unterstützt mehrere Display-Seiten, auf denen ein Verhalten beispielsweise seine Zustandswerte darstellen kann. Noch besser geeignet sind die LOG-Funktionen, die Informationen sowohl auf das Display als auch zum Simulator ausgeben können. Sie akzeptieren die printf()-Syntax. Mit ihnen ist auch eine Differenzierung verschiedener Log-Level möglich (LOG_DEBUG, LOG_INFO). Das für den jeweiligen Einsatzzweck passende Ziel dieser Ausgaben wählt man durch Entfernen der Kommentarzeichen vor dem zugehörigen #define in ct-bot.h.

Die Anzahl der Ebenen, aus denen ein komplexes Verhalten besteht, ist nicht begrenzt. Das kann man sich zu Nutze machen und die Funktion von Hilfsverhalten – vor der Verwendung in einem komplexen Verhalten – mit Hilfe von kleinen Testverhalten oder der Fernbedienung prüfen. Tut man dies nicht, ist es später oft schwer, den wahren Schuldigen für ein Fehlverhalten des Bots auszumachen.

Reality Check!

Wer möchte, kann das hier vorgestellte Verhalten auch mit einem realen Bot ausprobieren. Allerdings benötigt das recht viel Platz, wenn man ein einigermaßen großes Labyrinth aufbauen möchte. Doch das ist oftmals nicht das einzige Problem. Viele Sensorenwerte und auch das Fahrverhalten sind stark von der Umgebung abhängig. So können starke Sonneneinstrahlung, unebenes Gelände oder Teppiche dafür sorgen, dass der reale Roboter nicht so weit kommt, wie man es aus dem Simulator gewohnt ist. Häufige Fehlerquelle sind hier die Abstandssensoren. Man sollte also – wie bereits an andere Stelle der Artikelreihe [3] beschrieben – eine Kalibrierung vornehmen. Ebenfalls problemtisch ist ein zu dunkler Untergrund, den die Kantensensoren als Abgrund interpretieren. Eine Anpassung der Konstante BORDER_DANGEROUS in bot-lokal.h an die vorhandene Umgebung schafft Abhilfe.

Das hier vorgestellte Verhalten lässt viel Raum für eigene Verbesserungen. Wer möchte, kann sich beispielsweise daran versuchen, den eingangs erwähnten Pledge-Algorithmus oder einen der anderen Algorithmen für die Aufgabe zu implementieren. Auch die Kombination mehrerer Verhalten ist eine interessante Herausforderung. Ein Labyrinth, bei dem der Bot sein Ziel nur erreicht, wenn er eine Linie findet, dieser folgt und mit ihr einen sonst nicht überbrückbaren Engpass überwindet, wäre eine Idee dazu.

Damit der c't-Sim dem Bot ein geeignetes Labyrinth präsentieren kann, hat er einige Erweiterungen erfahren. Diese sind jedoch Thema eines der kommenden Artikel in der ct-Bot-Reihe.

Wer über eigene Verhalten, Erweiterungen am Framework oder Algorithmen diskutieren möchte, erreicht über die Mailingliste

Hilfsverhalten

Verhalten	Botenfunktion	Beschreibung
bot_turn_behaviour()	bot_turn()	dreht den Bot um einen Winkel [Grad]
bot_goto_behaviour()	bot_goto()	lässt den Bot fahren [Encoderschritte links/rechts]
bot_drive_distance_behaviour()	bot_drive_distance()	lässt den Bot fahren [cm, Kurvenfaktor, Geschwindigkeit]
bot_explore_behaviour()	bot_explore()	erkundet seine Umwelt [check-Funktion]
bot_do_slalom_behaviour()	bot_do_slalom()	fährt Slalom um Hindernisse
bot_check_wall_behaviour()	bot_check_wall()	prüft Abstand/Winkel zu parallel stehenden Hindernissen [Richtung]
bot_measure_angle_behaviour()	bot_measure_angle()	dreht sich, bis ein Hindernis im Sensorbereich auftaucht [Richtung, max. Entfernung]

andere Entwickler und auch die Autoren der Artikelreihe. Gut dokumentierte und getestete Verhalten bauen wir gerne in die Code-Basis ein oder veröffentlichen sie auf der Projektseite [1]. Dort finden sich auch Links auf Diskussionsforen sowie die Mailingliste, eine ausführliche FAQ und Coding-Richtlinien. Letztere enthalten auch Regeln bezüglich der Platzierung von Konstanten, die vor allem für die leichte Wartbarkeit des Codes wichtig sind. Patches, die diese Richtlinien nicht einhalten, kommen nicht in die gemeinsame Codebasis. (bbe)

Literatur

[1] Webseite zum c't-Bot-Projekt: www.heise.de/ct/ftp/projekte/ct-bot

[2] Benjamin Benz, Peter König, Lasse Schwarten, Drängelnde Spielgefährten, Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot, c't 5/06, S. 224

[3] Benjamin Benz, Nervensystem, Programmierung des c't-Bot von der Pike auf, c't 6/06, S. 264

[4] Christoph Grimmer, Hohe Schule, c't-Bots bewältigen komplexe Aufgaben, c't 7/06, S. 218

[5] http://www2.in.tu-clausthal.de/~hormann/teaching/ProSemWS04/PS.AlGeo.11.01.2005.a.pdf

Soft-Link 0610236

Wie sag ich es meinem Bot?

Ist die Planungsphase für ein neues Verhalten abgeschlossen, steht die Umsetzung des Algorithmus in ein Verhalten für das c't-Bot-Framework auf dem Plan. Die Datei botlogik.c enthält alle bereits zur Verfügung stehenden Routinen und ist auch der richtige Ort für eigenen Verhaltenscode.

Damit der c't-Bot einen Algorithmus ausführt, muss man ihn in ein Verhalten verpacken. Das Verhaltens-Framework übernimmt dann die Ausführung. Der Begriff „Verhalten" beschreibt die konkrete Implementation eines (Teil-)Algorithmus. Streng genommen besteht es aus einer Funktion, die durch Auslesen und Auswerten von Sensoren notwendige Modifikationen für die Aktuatorsteuerung ermittelt. Alle Verhaltensfunktionen folgen der Nomenklatur bot_XXX_behaviour().

Die Verwaltung von Verhaltensfunktionen erfolgt in einer priorisierten Liste, die bot_behave_init() aufbaut. In dieser Liste können Verhalten mit dem Status ACTIVE stehen, die bereits beim Einschalten des Bots ihre Arbeit aufnehmen, wie solche zur Gefahrenabwehr. Ein mit dem Status INACTIVE gekennzeichnetes Verhalten kommt nur zum Zug, wenn es explizit von einem anderen Verhalten oder der Fernbedienung aktiviert wird. Typische Vertreter dieser Spezies sind Hilfsverhalten wie bot_turn_behaviour(). Sie werden über eine Botenfunktion gestartet, deren Aufruf auf den ersten Blick wie der einer normalen Funktion aussieht und auch genauso einfach zu handhaben ist:

```
bot_turn(data,90);
```

Botenfunktionen heißen bot_XXX(). Die Variable data enthält neben internen Verwaltungsstrukturen des Framework auch einen Zeiger auf das Verhalten, das diese Botenfunktion aufruft. Damit ist sichergestellt, dass das Framework immer weiß, zu welchem Verhalten es nach dem Ende des Hilfsverhaltens zurückkehren soll. Die weiteren Parameter sind von Botenfunktion zu Botenfunktion verschieden. Da es vorkommen kann, dass mehrere Verhalten dasselbe Hilfsverhalten aufrufen, entscheidet der Bote, ob er einen laufenden Auftrag unterbrechen will. Der Parameter NOOVERRIDE/OVERRIDE der Routine switch_to_behaviour kontrolliert, ob der zweite Aufruf den ersten überschreiben soll. In diesem Fall bekommt das Verhalten, welches den ersten Auftrag ausgelöst hat, die Nachricht SUBFAIL – die es selbst auswerten muss. Wie das aussehen kann, zeigt der Code-Abschnitt etwas weiter unten.

Der Bote für bot_turn_behaviour ist freundlich und überschreibt nicht:

```
void bot_turn(Behaviour_t* caller,int16 degrees){
  if(degrees < 0)
    turn_direction = -1;
  else
    turn_direction = 1;
  turn_targetR=(degrees*ANGLE_CONSTANT)/360;
  turn_targetR+=sensEncR;
  ...
  switch_to_behaviour(caller,
                      bot_turn_behaviour,NOOVERRIDE);
}
```

Er übergibt außerdem einige Startwerte an das Hilfsverhalten. Zu Beginn der Datei botlogic.c befinden sich dazu bereits einige globale Variablen wie beispielsweise turn_direction. Weitere Details zum Aufbau des Framework finden sich in den Artikeln [2] und [3]. Das Einfügen eines Verhaltens in die Liste erfolgt über:

```
insert_behaviour_to_list(&behaviour,
    new_behaviour(200,bot_turn_behaviour,INACTIVE));
```

Das Aufwecken von Verhalten mit dem Status INACTIVE übernehmen activateBehaviour() oder switch_to_behaviour(). Der Unterschied zwischen diesen beiden Funktionen liegt darin, dass activateBehaviour() das gewünschte Verhalten zwar aktiviert, aber das aufrufende Verhalten nicht beeinflusst. Meist ist aber die Funktion switch_to_behaviour() sinnvoller, denn dann wartet das aufrufende Verhalten, bis die Aufgabe komplett erledigt ist. Unabhängig davon, wer ein Verhalten aktiviert, kommt es erst zum Zug, wenn es von seiner Priorität her an der Reihe ist und kein anderes, höher priorisiertes Verhalten eine Geschwindigkeitsänderung veranlasst hat.

Verhaltensweisen auf höherer Abstraktionsebene wie bot_do_slalom_behaviour() bilden neben denen zur Gefahrenerkenung und denen für Hilfsaufgaben die dritte Variante der Verhaltensfunktionen. Sie bedienen sich meist mehrerer Hilfsverhalten, um komplexere Aufgaben zu lösen. So nutzt bot_solve_maze_behaviour() für die Prüfung des Abstands zur Wand bot_check_wall_behaviour() und bot_measure_angle_behaviour() für die Ermittlung des Drehwinkels zu einer Ecke.

Verhalten kriegen Zustände

Um das Labyrinth zu durchqueren, durchläuft das Hauptverhalten des Höhlenforschers mehrere Zustände in einem Automaten. Jeder Zustand löst eine oder mehrere der oben besprochenen Aufgaben. Konstanten, die den jeweiligen Zuständen verständliche Namen zuordnen, verbessern die Lesbarkeit:

```
void bot_drive_square_behaviour(Behaviour_t* data){
#define STATE_FORWARD 0
#define STATE_TURN 1
#define STATE_INTERRUPTED 2
static uint8 state = STATE_FORWARD;

if (data->subResult == SUBFAIL)
  state= STATE_INTERRUPTED;
switch (state) {
  case STATE_FORWARD:
    bot_drive_distance(data,0,BOT_SPEED_NORMAL,20);
    state = STATE_TURN;
  break;
  case STATE_TURN:
    bot_turn(data,90);
    state=STATE_FORWARD;
  break;
  default:
    return_from_behaviour(data);
  break;
}
```

Eine statische Variable vom Typ int8 enthält jeweils den aktuellen Zustand und sichert ihn über mehrere Aufrufe hinweg. Die switch-Anweisung führt nur den jeweils notwendigen Verarbeitungsschritt aus und gibt die Kontrolle dann zurück an das Verhaltens-Framework. Ist die Aufgabe abgearbeitet, muss sich das Verhalten ordnungsgemäß beenden, indem es return_from_behaviour() aufruft.

c't