Simulationsmodell zur Bewertung von Schedulingalgorithmen für einen mobilen Roboter

Lars Gendner
Fachhochschule Hamburg
Fachbereich Elektrotechnik und Informatik
Studiengang Softwaretechnik

1. April 1997

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein häufiges Problem in der Robotik ist die Betrauung eines Roboters mit einer Aufgabe. Was geschieht jedoch in einer Situation, in der ein und derselbe Roboter mehrere Aufgaben gleichzeitig gestellt bekommt? Es taucht die Frage auf, welche Aufgabe wann ausgeführt werden soll. Mit dieser Problematik befaßt sich die Klasse der Schedulingalgorithmen.

In dieser Arbeit soll ein System entwickelt werden, mit dem verschiedene Schedulingstrategien für Roboter bewertet und verglichen werden können. Dazu muß die Möglichkeit bestehen, verschiedene Schedulingstrategien einzubinden und während der Simulation passende statistische Werte auszugeben. Diese Werte können mit Verfahren aus der mathematischen Statistik ausgewertet oder mit der geeigneten Software (z.B. Gnuplot) visualisiert werden.

Ein typischer Bereich, in dem mehrere Aufgaben gleichzeitig anfallen, ist die Gastronomie. In gastronomischen Betrieben kommen und gehen Gäste und erwarten, daß ihre Wünsche zügig erfüllt werden. Dies stellt hohe Anforderungen an das Bedienpersonal.

Grundidee dieser Arbeit ist, einen Roboter mit der Aufgabe der Kundenbetreuung in einem solchen Betrieb zu betrauen.

Deswegen wird das verwendete Modell einen Gastronomiebetrieb, z.B. ein Restaurant, simulieren. Hier soll der Roboter auf Bestellungen von Gästen reagieren, indem er sie annimmt und an Küche oder Theke weiterreicht. Stehen die angeforderten Speisen und Getränke bereit, soll er sie den Gästen an den Tisch bringen. Schließlich muß abgeräumt und kassiert werden.

Das System muß fähig sein, parallele Handlungen und asynchrone Ereignisse zu simulieren. Es muß ein virtueller Roboter entworfen werden, der die anfallenden Aufgaben erfüllen kann. Andererseits sind Gäste zu simulieren, die diese Aufgaben auslösen.

Kapitel 2

Das Simulationsmodell

2.1 Invarianter Teil der Simulation

Der statische Teil des Simulationsmodells modelliert die Aspekte des Arbeitsbereiches, die der Roboter nicht durch Aktionen manipulieren kann. Der Arbeitsbereich wird durch einen geometrischen Graphen repräsentiert. Die Ecken dieses Graphen bilden die Anlaufpunkte des Roboters, welcher auf den Kanten entlangfährt. Eine Ecke gehört zu einem der folgenden Typen:

- normale Ecke
- Tisch
- Essensausgabe
- Getränkeausgabe
- Entsorgungseinheit

Abbildung 2.1 zeigt zur Veranschaulichung die Darstellung eines möglichen Arbeitsbereiches. Essens- und Getränkeausgabe (*kitchen*, *bar*) dienen dazu, auf Bestellung Objekte zu erzeugen. Die Entsorgungseinheit (*backyard*) entnimmt dem System alle Objekte, die an sie übergeben werden. Es gibt nur jeweils eine Essensausgabe, eine Getränkeausgabe und eine Entsorgungseinheit im Arbeitsbereich, die Anzahl der Tische (*t1a*, *t1b*, *t1c*, *t2a*, *t2b*, *t2c*) und der normalen Ecken (*crossing1*, *crossing2*, *area1*, *area2*) ist beliebig.

Der Aufbau des Graphen ist dem System bekannt und während der Laufzeit keinen Veränderungen unterworfen. Der Roboter kann den statischen Teil nicht durch seine Handlungen beeinflussen.

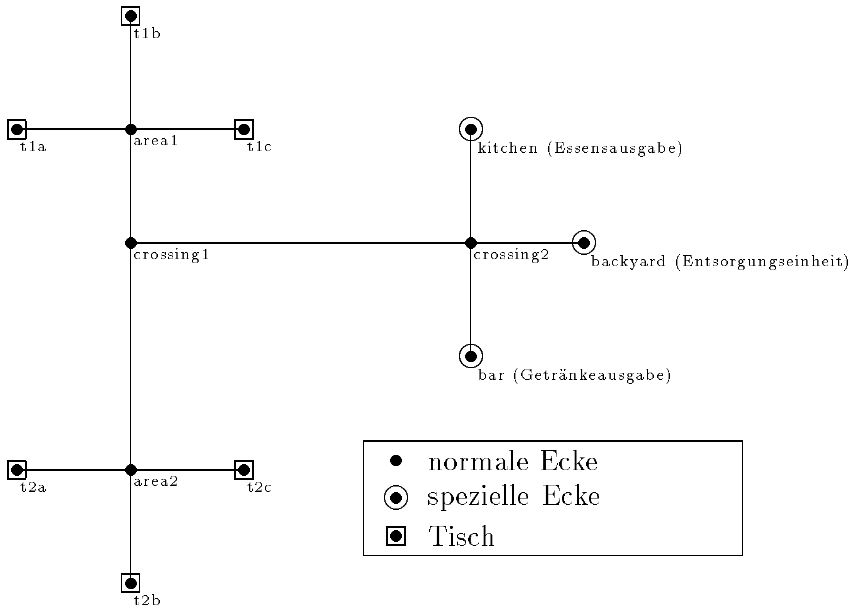

Abbildung 2.1: Beispiel für einen Arbeitsbereich

2.2 Dynamischer Teil der Simulation

Der dynamische Teil umfaßt die Aspekte des Systems, die während der Laufzeit Änderungen unterliegen. Dies sind die Zustände des Roboters, der Gäste und der verschiedenen Objekte.

Der Zustand des Roboters ist definiert durch seine Position, seine Blickrichtung, von ihm transportierte Objekte und die ihm augenblicklich zugeteilte Aufgabe. Die Position ist beschrieben durch den Namen der Ecke innerhalb des Graphen. Die Blickrichtung bezeichnet die Richtung, in der sich der Roboter bewegt, wenn er vorwärts fährt. Sie ist angegeben in einem Winkel zwischen 0° und 360°, wobei 0° Norden bedeutet und der Winkel im Uhrzeigersinn zunimmt.

Bestellungen werden im vorhandenen Modell von Gästen erzeugt, die in variablen Zeitabständen eintreffen und sich an einen freien Tisch setzen.Um sie aufzunehmen, muß sich der Roboter zu dem jeweiligen Tisch begeben. Er analysiert die Bestellung und gibt sie an die Essens- und die Getränkeausgabe weiter.

Objekte werden nur an der Essens- und an der Getränkeausgabe erzeugt. Geht eine Bestellung für ein Objekt ein, steht es nach Ablauf einer zufälig ermittelten Zeit zur Abholung bereit. Wird ein Objekt auf einem Tisch abgestellt, an dem es bestellt wurde, wird es innerhalb einer ebenfalls zufälligen Zeit konsumiert. Zurück bleibt Geschirr, das abgeräumt und entsorgt werden muß. Für diesen Zweck ist die Entsorgungseinheit vorgesehen. Sie entnimmt dem System jedes Objekt, das ihr übergeben wird. Den dynamischen Bereich kann der Roboter durch sein Verhalten direkt oder indirekt beeinflussen.

2.3 Realisierung der Simulation

2.3.1 Die Ereignisliste

Ein Charakteristikum des gegebenen Systems ist, daß sich dessen Zustand nur zu diskreten Zeitpunkten verändert. Diese Zustandsänderungen werden durch Ereignisse herbeigeführt. Ein solches System nennt man zeitdiskret.

Zeitdiskrete Systeme können ereignisorientiert beschrieben und simuliert werden. Es wird dabei eine Ereignisliste verwendet. Die folgende Beschreibung ist an [Siegert, 1991] angelehnt.

Ein Ereignis wird durch das Tupel Ereignisart e.a und Eintreffzeitpunkt e.t beschrieben. Es wird ein ADT Ereignisliste definiert, der folgende Schnittstelle bereitstellt:

- **queueEvent(e)** — Das Ereignis e wird in die Ereignisliste $\mathcal{E}$ eingetragen.
- **dequeueEvent(e)** — Das Ereignis e wird aus der Ereignisliste $\mathcal{E}$ gelöscht. In der vorliegenden Implementation des Systems wird diese Operation nur vom System ausgeführt. Dies geschieht für jedes eingetroffene Ereignis.
- **nextEvent** — Diese Funktion liefert das nächste anstehende Ereignis. Dieses ist definiert als
$$e_j \in \mathcal{E}, \forall e_i \in \mathcal{E} : e_i.t \geq e_j.t$$
Wie bereits erwähnt, wird e_j aus der Ereignisliste gelöscht. Es sei darauf hingewiesen, daß jedes Ereignis eine eindeutige Identität hat. Zwei Ereignisse e_i, e_j mit $e_i.a = e_j.a \wedge e_i.t = e_j.t$ können durchaus zwei Instanzen derselben Ereignisart sein, die zeitgleich eintreffen.

  In diesem Fall treffen die Ereignisse in der Reihenfolge ihrer Eintragung in die Ereignisliste ein. Es wird angenommen, daß sich die Eintreffzeitpunkte in einer Größenordnung unterscheiden, die nicht im Bereich der zeitlichen Granularität der Simulation liegt.

Während der Simulation wird das System jeweils für die Zeitspanne bis zum nächsten Ereignis schlafengelegt. Nun findet dieses Ereignis statt, wobei es im Allgemeinen neue Ereignisse auslöst, die dann wiederum in die Ereignisliste eingetragen werden.

2.3.2 Wiederkehrende Ereignisse

Es gibt Ereignisse, die in regelmäßigen Zeitabständen eintreffen. Um diesen Sachverhalt abzubilden, wurde die Schnittstelle der Ereignisliste erweitert. Für ein Ereignis kann bei der Eintragung in die Ereignisliste optional angegeben werden, wie oft es eintrifft und wie die Zeiträume zwischen den einzelnen Ereignisinstanzen berechnet werden[1].

Die Dauer dieser Intervalle kann im einfachsten Fall durch eine Konstante gegeben sein. Häufig ist jedoch gefordert, daß sie einer Zufallsgröße entspricht, die eine bestimmte Verteilungsfunktion besitzt.

Um dies zu erreichen, müssen entsprechend verteilte Zufallszahlen erzeugt werden. Da von den meisten gängigen Programmiersprachen nur die Erzeugung gleichverteilter Zufallszahlen unterstützt wird, wird im folgenden Abschnitt ein allgemeines Verfahren zur Erzeugung beliebig verteilter Zufallszahlen aus Realisierungen einer gleichverteilten Größe eingeführt.

2.3.3 Inverse Transformation

Das hier vorgestellte Verfahren der inversen Transformation ist [Siegert, 1991] entnommen. Dort kann auch die Herleitung des Verfahrens nachgelesen werden.

Ziel ist es, Realisierungen einer Zufallsgröße $\underline{u}$ zu erzeugen, deren Dichte $g(u)$ bzw. Verteilungsfunktion $G(u)$ vorgegeben ist. Sei $\underline{r}$ eine in $[0,1[$ gleichverteilte Zufallszahl, so kann mittels der Beziehung

$$u = G^{-1}(r) \tag{2.1}$$

aus einer Realisierung r der Zufallsgröße $\underline{r}$ eine Realisierung u der zufälligen Größe $\underline{u}$ erzeugt werden.

2.3.4 Erzeugung exponentialverteilter Zufallszahlen

Als Anwendungsbeispiel sollen hier die Realisierungen x einer exponentialverteilten [2] Zufallsgröße $\underline{x}$ erzeugt werden. Die Exponentialverteilung hat die Vertei-

[1] Ereignisinstanz bezeichnet hier ein einmaliges Eintreffen eines wiederkehrenden Ereignisses.

[2] bei [Siegert, 1991]: *negativ exponentiell verteilt*

lungsfunktion

$$F(x) = \begin{cases} 0 & \textit{für} \quad x < 0 \\ 1 - e^{-\alpha x} & \textit{für} \quad x \geq 0 \end{cases} \tag{2.2}$$

deren Umkehrfunktion

$$F^{-1}(y) = -\frac{1}{\alpha} * \ln(1 - y) \tag{2.3}$$

ist. Sei r die Realisierung einer in $[0, 1[$ gleichverteilten Zufallsgröße, so folgt aus den Formeln (2.1) und (2.3), daß über die Beziehung

$$u = F^{-1}(r) = -\frac{1}{\alpha} * \ln(1 - r) \tag{2.4}$$

Realisierungen u der exponentialverteilten Zufallsgröße $\underline{u}$ erzeugt werden können.

Kapitel 3

Die Systemkomponenten

Die Komponenten des Systems werden in zwei Klassen eingeteilt. Als aktive Systemkomponenten werden die Objekte bezeichnet, die durch ihr Verhalten den dynamischen Teil der Simulation beeinflussen. Sie sind speziell dadurch erkennbar, daß sie Nachrichten an andere Komponenten verschicken oder von diesen erhalten[1].

Passive Komponenten werden durch die aktiven Komponenten manipuliert. Sie dienen als Datenspeicher und haben kein eigenes Verhalten.

So sind z.B. normale Ecken passive Komponenten, da sie nur dazu benutzt werden, die Struktur des Arbeitsbereiches abzubilden oder den Aufenthaltsort von Objekten festzuhalten. Der Roboter hingegen ist aktiv, da er beispielsweise auf eine Nachricht des Schedulers hin eine Aufgabe ausführt und sich nach deren Beendigung als verfügbar zurückmeldet.

3.1 Kommunikation zwischen den Komponenten

3.1.1 Der Nachrichtenmechanismus

Die Kommunikation zwischen den Komponenten des Systems wird mittels eines einfachen Mechanismus zum Senden und Empfangen von Nachrichten realisiert. Jede abgesetzte Nachricht wird mit ihrem jeweiligen Sender und Empfänger gespeichert. Liest ein Empfänger eine Nachricht, so wird diese als gelesen markiert. Für den Empfänger sind nur die ungelesenen, an ihn adressierten Nachrichten

[1] Das Nachrichtensystem wird in Abschnitt 3.1 erläutert.

sichtbar. Somit stellt der Mechanismus sicher, daß jede Nachricht nur einmal zugestellt wird.

Jede Systemkomponente, die Nachrichten erhalten soll, muß als Empfänger registriert werden. Nachrichten an unregistrierte Empfänger werden erkannt und abgefangen. Beim Verlassen des Systems sollte jeder Empfänger seine Registrierung aufheben.

Das Senden ist grundsätzlich jeder Komponente erlaubt, wird jedoch dadurch eingeschränkt, daß die registrierten Empfänger meist nur bestimmte Nachrichten von bestimmten Sendern erwarten und alle anderen ignorieren.

Das System gewährleistet die Übermittlung jeder Nachricht innerhalb der nächsten fünf Sekunden Simulationszeit.

In variablen Abständen wird es jedem registrierten Empfänger ermöglicht, seine Nachrichten zu lesen. Dies geschieht, indem der bei der Registrierung angegebene Handler aufgerufen wird. Im Normalfall liest dieser Handler die anstehenden Nachrichten und löst die adäquaten Reaktionen aus. Damit ein Handler gegebenenfalls für mehrere Empfänger verwendet werden kann, wird bei seinem Aufruf der Name des Empfängers mit übergeben. Es wird darauf vertraut, daß nur die Nachrichten gelesen werden, die für den jeweiligen Empfänger bestimmt sind.

3.1.2 Sinn und Zweck der Nachrichten

Die Entscheidung, Nachrichten als Kommunikationsmittel zu wählen, bot sich aus verschiedenen Gründen an. Zum einen ist so klar erkennbar, an welchen Stellen Komponenten miteinander kommunizieren. Ein anderer Vorteil ist die Unidirektionalität von Nachrichten. Durch sie ist die Datenflußrichtung eindeutig vom Sender zum Empfänger festgelegt.

Ein anderes wichtiges Argument, gerade in Bezug auf den temporalen Charakter der Simulation, ist die zeitliche Entkopplung von Sender und Empfänger. Der Sender kann zwar den Zeitpunkt der Nachrichtenabsendung wählen, hat jedoch keinen Einfluß darauf, wann diese vom Empfänger bearbeitet wird.

Da der Absender einer Nachricht bekannt ist, kann zudem auf Empfängerseite eine Zugriffskontrolle geschehen, wie sie bei anderen Mechanismen nicht möglich wäre.

3.2 Beschreibung der einzelnen Komponenten

3.2.1 Der Roboter

Basisfunktionalität des Roboters

Die Basisfunktionen des Roboters sind die atomaren Handlungen, die von ihm durchgeführt werden können. Sie dienen als mögliche Schnittstelle zu anderen Systemen, z.B. zu einem wirklich vorhandenen Roboter, der die Simulation ersetzt. Diese Tätigkeiten können nur sequentiell ausgeführt werden.

Die Kosten einer Handlung werden an den für ihre Ausführung benötigten Zeiteinheiten gemessen. Sie sind absolut und nicht von der Position des Roboters abhängig.

- Bewegung
  - **Vorwärts/rückwärts fahren** — Der Roboter bewegt sich in die angegebene Richtung, vorausgesetzt dort befindet sich eine Kante. Die Bewegung endet an der nächsten Ecke. Die Kosten für die Durchführung dieser Handlung errechnen sich aus der Fahrgeschwindigkeit des Roboters und der Länge der Kante.
  - **Links/rechts drehen** — Der Roboter dreht sich um den spezifizierten Winkel in die angegebene Richtung. Eine Drehung nach rechts bedeutet von oben gesehen im Uhrzeigersinn, eine Linksdrehung entsprechend gegen den Uhrzeigersinn. Die Einheit des Winkels ist Grad. Hier errechnen sich die Kosten aus der Drehgeschwindigkeit und dem Drehwinkel.
- Kommunikation
  - **Bestellung entgegennehmen** — Diese Aktion wird immer auf die Ecke angewendet, auf der sich der Roboter befindet. Diese muß ein Tisch sein, an dem sich ein Gast befindet.

    Da der Gast bereits beim Platznehmen seine Bestellung zusammenstellt, muß dieser Prozeß nicht erst durch eine Nachricht initiiert werden. Der Roboter kann sofort eine Liste von Getränken und Speisen erhalten. Er analysiert Liste und speichert getrennt nach Getränken und Speisen
    * eine Liste der bestellten Getränke bzw. Speisen,
    * den Gast, der sie bestellt hat.

In der vorliegenden Implementation werden die Datenstrukturen mit diesen Informationen als normale Objekte behandelt, die aufgenommen und abgelegt werden können. Dies umgeht die Notwendigkeit eines speziellen Mechanismus zum Informationsaustausch zwischen dem Roboter und der Essens- oder Getränkeausgabe. Im folgenden werden diese Objekte als Getränke- und Essensbestellungen bezeichnet.

Die Kosten für diese Aktion sind immer gleich und werden mittels einer Konstanten festgelegt.

  - **Kassieren** — Das Kassieren eines Gastes ist nur möglich, wenn sich dieser mit dem Roboter am selben Ort befindet. Es wird der Rechnungsbetrag ermittelt[2] und dem Gast mitgeteilt. Dieser antwortet mit einer Nachricht, die den passenden Betrag beinhaltet[3].

- Manipulation

  - **Objekt nehmen** — Diese Aktion entfernt das angegebene Objekt von der Ecke, auf der der Roboter steht, und nimmt es in dessen Inventar auf. Befindet sich dieses Objekt nicht dort, schlägt die Aktion fehl. Im erfolgreichen Fall kostet diese Handlung einen festen Betrag an Zeiteinheiten. Der Roboter kann eine unbegrenzte Anzahl von Objekten gleichzeitig tragen.
  - **Objekt abstellen** — Ist das angegebene Objekt Teil des Roboterinventars, wird es auf die Ecke gestellt, die der Standort des Roboters ist. Die Kosten sind im Falle der erfolgreichen Ausführung abermals konstant.

Bewegungsplanung

Bevor Aufgaben auf Schedulingebene realisierbar sind, muß der Roboter zuerst zu gezielter Bewegung im Arbeitsbereich fähig sein. Dies wird mittels eines Kommandos erreicht, das in seiner Komplexität zwischen der Basisfunktionalität und der Schedulingebene liegt.

- **Zu einer Ecke fahren** — Dieses Kommando umfaßt die gesamte Routen- und Bewegungsplanung. Sie ermöglicht es, dem Roboter ein Ziel zu nennen, ohne von dessen Position zum Zeitpunkt der Aufgabenausführung abhängig zu sein. Die eigentliche Planung, mit welchen Basisbewegungen der Roboter

[2] Zur Zeit fällt die Rechnung für jeden Gast recht niedrig aus: Das System berechnet grundsätzlich den Rechnungsbetrag Null.

[3] An dieser Stelle ist das System noch ausbaufähig. Das Geld könnte vom Gast als ein oder mehrere Objekte auf dem Tisch plaziert werden. Der Roboter müßte dies erkennen und an sich nehmen. Ein weiterer Aspekt könnte das Anfallen von Trink- bzw. Wechselgeld sein.

dieses Ziel erreicht, geschieht erst im Moment der Ausführung. Mit diesem Instrument ist die Formulierung höhergestellter Aufgaben möglich.

Aufgaben auf Schedulingebene

Auf der Ebene der Basisfunktionalität ist noch kein Scheduling zu realisieren. Um eine sinnvolle Problemlösungssemantik zu erhalten, müssen die atomaren Handlungen zu komplexeren Aufgaben aneinandergereiht werden. Welche Handlungen zur Erfüllung einer Aufgabe nötig sind, hängt von ihrer Charakteristik und der Position des Roboters ab.

Ist die Aufgabe z.B.: "stelle Objekt A auf Ecke B“, so besteht ein Unterschied zwischen einer Situation, in der sich der Roboter bereits auf Ecke B befindet, und einer Situation, in der er erst dorthin gelangen muß.

Es folgt eine Liste der Aufgaben, die dem Scheduler zur Einplanung übergeben werden. Für jede sind die Basisfunktionen angegeben, aus denen sie sich zusammensetzt.

- **Objekt von einer Ecke aufnehmen** —
  1. Zur Ecke fahren
  2. Objekt nehmen
- **Objekt auf einer Ecke ablegen** —
  1. Zur Ecke fahren
  2. Objekt abstellen
- **Bestellung eines Gastes aufnehmen** —
  1. Zur Ecke des Gastes fahren
  2. Bestellung aufnehmen
- **Objekt zu einem Gast bringen** —
  1. Zur Ecke des Gastes fahren
  2. Objekt abstellen
  3. Den Gast benachrichtigen
- **Getränke- oder Essensbestellung auf einer Ecke ablegen** —
  1. Zur Ecke fahren
  2. Objekt abstellen

3. Die Ecke benachrichtigen

- **Gast kassieren** —
  1. Zur Ecke fahren
  2. Kassieren
- **Tisch abräumen** —
  1. Zum Tisch fahren
  2. Alle Objekte nehmen
  3. Scheduler benachrichtigen
- **Objekte entsorgen** —
  1. Zur Entsorgungseinheit fahren
  2. Objekte abstellen
  3. Entsorgungseinheit benachrichtigen

3.2.2 Die Essensausgabe

Eine Ecke des Arbeitsbereiches muß als Essensausgabe fungieren. Ihre Aufgabe besteht darin, auf eine Nachricht des Roboters hin alle auf ihr abgelegten Essensbestellungen zu analysieren und die darauf enthaltenen Speisen herzustellen. Sie werden nach einer festgelegten Herstellzeit auf der Essensausgabe plaziert. Gleichzeitig wird der Scheduler benachrichtigt, daß eine Speise bereitsteht. Diese Nachricht enthält auch einen der Bestellung entnommenen Verweis auf den Gast, der diese aufgegeben hat. So ist es für den Scheduler nicht notwendig, darüber Buch zu führen, welcher Gast welche Bestellung aufgegeben hat.

3.2.3 Die Getränkeausgabe

Diese Komponente verhält sich äquivalent zur Essensausgabe. Der einzige naheliegende Unterschied besteht darin, daß an der Getränkeausgabe nur Getränkebestellungen bearbeitet werden.

3.2.4 Die Entsorgungseinheit

Auch für die Entsorgungseinheit gilt, daß es im Arbeitsbereich genau eine dafür bestimmte Ecke geben muß. Auf eine entsprechende Nachricht vom Roboter reagiert sie mit der dauerhaften Entnahme aller auf ihr abgestellten Objekte aus dem System. Da der Zeitpunkt keinerlei Einfluß auf den Simulationsverlauf hat, geschieht die Entsorgung sofort ohne simulierte Verzögerung. Ebenso wird keine Nachricht versendet.

3.2.5 Die Rezeption

Die Rezeption ist keine Örtlichkeit sondern ein virtueller Vermittler zwischen den Gästen und dem eigentlichen System. Ihre Hauptaufgabe ist die Verwaltung der freien Tische.

Tritt ein Gast in das System ein, signalisiert er der Rezeption, daß er einen Tisch sucht. Diese wählt nach dem Zufallsprinzip einen freien Tisch, den sie dem Gast zuteilt. Durch eine Nachricht an den Gast wird dieser aufgefordert, seine Datenbestände entsprechend anzugleichen. Ist kein freier Tisch verfügbar, wird der Gast in eine Warteschlange eingereiht. Wird ein Tisch frei und befinden sich Gäste in der Warteschlange, so wird dieser Tisch dem ersten Gast in der Warteschlange zugeteilt.

3.2.6 Die Gäste

Die effektive Erfüllung der Bedürfnisse der Gäste ist die Hauptaufgabe des Systems. Der Zeitraum zwischen zwei aufeinanderfolgend eintretenden Gästen ist durch die Zufallsvariable $\underline{x}$ gegeben. $\underline{x}$ genügt der *Exponentialverteilung mit dem Parameter* α_x[4]:

$$F(x) = \begin{cases} 0 & \mathit{für} \quad x < 0 \\ 1 - e^{-\alpha_x x} & \mathit{für} \quad x \geq 0 \end{cases} \tag{3.1}$$

$\frac{1}{\alpha_x}$ ist hierbei die mittlere Zeit, die zwischen zwei Gästebesuchen vergeht. Daraus folgt unmittelbar, daß α_x die mittlere Eintreffrate in Gästen pro Zeiteinheit darstellt[5]. Folgende Gründe sprechen für die Verwendung der Exponentialverteilung:

[4] Für ein Verfahren zur Erzeugung exponentialverteilter Zufallszahlen siehe Abschnitt 2.3.4. Für weitere Informationen über die Exponentialverteilung siehe z.B. [Storm, 1988]

[5] Eine Zeiteinheit entspricht einer Sekunde (Simulationszeit). Die mittlere Eintreffrate in Gästen pro Stunde ergibt sich zu $\alpha_x * 3600$

- Sie ist *gedächtnislos* (siehe [Siegert, 1991]), d.h. daß die Zeit bis zum Eintreffen des nächsten Kunden im Mittel immer $\frac{1}{\alpha_x}$ beträgt, egal wie lange die Ankunft des letzten Kunden zurückliegt,
- über den Parameter α kann direkt der Erwartungswert gewählt werden,
- laut [Siegert, 1991] werden Ankunftsprozesse im Allgemeinen mithilfe der Exponentialverteilung simuliert.

Sobald ein Gast in das System eingetreten ist, setzt er sich an einen ihm zugeteilten Tisch. Er wählt dann ein Getränk und/oder ein Gericht von der Speisekarte und gibt seine Bestellung auf.

Nach Erhalt einer Bestellung konsumiert der Gast innerhalb einer durch die Zufallsvariable $\underline{y}$ bestimmten Zeitspanne das Gericht und läßt nur die Überreste auf dem Tisch zurück. $\underline{y}$ genügt ebenfalls der Exponentialverteilung ist für alle Gerichte mit demselben α_y parametrisiert. $\frac{1}{\alpha_y}$ bezeichnet hier die durchschnittliche Zeit, in der ein Gericht konsumiert wird.

Wurde das letzte bestellte Gericht konsumiert, so verlangt der Gast nach einer durch die exponentialverteilte Zufallsvariable $\underline{z}$ bestimmten Dauer die Rechnung. Dies geschieht durch eine Nachricht an die Rezeption. Analog zu den obigen Verteilungen ist der Mittelwert auch hier gegeben durch $\frac{1}{\alpha_z}$.

Bei Erhalt der Rechnung wird diese bezahlt[6] und auf die Verabschiedungsnachricht vom Roboter gewartet. Nachdem die Rezeption über den freiwerdenden Tisch benachrichtigt wurde, verläßt der Gast das System.

3.2.7 Der Scheduler

Der Scheduler ist die Systemkomponente, der die Aufgabenverwaltung zukommt. Er bildet die Schnittstelle zwischen dem System und der eigentlichen Implementation eines Scheduling-Algorithmus. Bei der Initialisierung muß bestimmt werden, welcher Algorithmus verwendet werden soll. Jeder verwendete Algorithmus muß der folgenden Schnittstellendefinition genügen.

- **Initialisierung** — Diese Funktion wird nur einmal zum Systemstart aufgerufen. In ihr sollte der Scheduler alle notwendigen Initialisierungen durchführen.
- **Aufgabe einplanen** — Hiermit wird dem Scheduler eine neue Aufgabe zur Einplanung übergeben.

[6] Die Gäste haben zur Zeit immer den passenden Betrag zur Hand. Bezahlung durch Tellerwaschen muß also nicht in Erwägung gezogen werden.

- **Nächste Aufgabe** — Diese Anfrage sollte die nächste anstehende Aufgabe liefern.

- **Aufgabe entfernen** — Mit dieser Anfrage wird der Scheduler dazu aufgefordert, die spezifizierte Aufgabe aus der Planung zu entfernen.

Die Verkettung von Aufgaben

Aufgabenverkettung wird dann notwendig, wenn die Erfüllung einer Aufgabe eine neue Aufgabe aufwirft. Dies ist zur Zeit nur in einer Situation der Fall. Liegt an der Essens- oder Getränkeausgabe ein Objekt für einen Gast bereit, so muß es von dort geholt und zum Gast gebracht werden.

Es ist jedoch nicht sinnvoll, die Aufgabe einzuplanen, dem Gast das Objekt zu bringen, solange der Roboter dieses noch nicht in seinem Inventar hat. Ein Scheduling-Algorithmus mit zuwenig Intelligenz würde eventuell versuchen, dem Gast das Objekt bringen zu lassen, bevor es aufgenommen wurde.

Um dieses Problem zu lösen, bietet Aufgabenverkettung die Möglichkeit, dem Scheduler eine Liste von Aufgaben zu übergeben, wobei gewährleistet ist, daß ein Element dieser Liste erst eingeplant wird, wenn alle in der Liste vorstehenden Elemente erfüllt sind.

Das Zuteilen von Aufgaben

Jedesmal, wenn der Roboter seine Verfügbarkeit signalisiert oder eine neue Aufgabe eingeplant wurde, wird versucht, dem Roboter eine Aufgabe zuzuteilen. Dies ist nur erfolgreich, wenn der Roboter verfügbar und eine Aufgabe zu erledigen ist.In diesem Fall wird dem Roboter die betreffende Aufgabe übergeben.

In der vorliegenden Implementation ist der Scheduler nur zur Verwaltung eines Roboters fähig. Bei mehreren Robotern als mögliche Instanzen zur Durchführung einer Aufgabe müßte die Schnittstelle zum Algorithmus so erweitert werden, daß dieser den für die Ausführung gewählten Roboter spezifizieren kann. Da die Bewertung der Kosten einer Aufgabe relativ zum ausführenden Roboter von Algorithmus zu Algorithmus stark variieren kann, sollte die Verantwortung für die Wahl des Roboters bei dem jeweiligen Scheduling-Algorithmus liegen.

Eine Aufgabe gilt als erledigt, wenn dies vom Roboter gemeldet wird. Sie wird dann aus der Planung genommen.

3.3 Änderbare Simulationsparameter

Das Verhalten der Simulation wird durch folgende Parameter bestimmt. Durch Anpassung dieser Werte kann auf die Dynamik des Systems Einfluß genommen werden.

3.3.1 Fest eingestellte Werte

Die folgenden Parameter sind fest im Programm vorgegeben. Die jeweils eingestellten Werte stehen in Klammern hinter der Beschreibung.

- Mittlere Herstellzeit für Speisen (10 min.),
- mittlere Herstellzeit für Getränke (3 min.),
- mittlere Zeit für die Konsumption von Speisen und Getränken (15 min.),
- mittlere Zeit, die zwischen der Konsumption des letzten Gerichts und dem Verlangen nach der Rechnung liegt (5 min.),
- Drehgeschwindigkeit des Roboters in Grad/Sekunde (45° /1 sec.),
- Fahrgeschwindigkeit des Roboters in Längeneinheiten/Sek. (0.1 LE/1 sec.),
- notwendige Zeit für die Aufnahme von Bestellungen (1 min.).

3.3.2 Initialisierungsparameter

Es folgen die Werte, die bei Initialisierung des Systems angegeben werden müssen.

- Gesamtanzahl der zu erzeugenden Gäste,
- mittlere Eintreffrate der Gäste in Gästen/Stunde,
- Anzahl der anfangs in der Warteschleife befindlichen Gäste,
- zu verwendender Scheduler,
- Definition des Arbeitsbereiches,
- Startposition des Roboters,
- anfängliche Blickrichtung des Roboters
- Skalierung der Simulationszeit in Sekunden/Simulationssekunde

3.4 Verfügbare Statistikwerte

Das System sammelt während der Simulation einige statistische Werte. Diese können dazu dienen, verschiedene Schedulingstrategien zu bewerten und zu vergleichen. Folgende Werte sind verfügbar:

- Die bisher simulierte Zeitspanne,
- Anzahl der Gäste, die das System betreten haben,
- Anzahl der Gäste, die das System bedient verlassen haben,
- der mittlere Zeitraum zwischen zwei hintereinander eintretenden Gästen[7],
- der Weg, der im Mittel für die Bedienung eines Gastes zurückgelegt wurde.

Ursprünglich sollten während der Simulation noch andere Werte für eine mögliche Auswertung gesammelt werden. Hiervon wurde aus Performancegründen abgesehen.

[7] Dieser Wert beschreibt den tatsächlichen und nicht den als Parameter festgelegten Zeitraum

Kapitel 4

Von der Aufgabe zum Bewegungsplan

Bisher wurde nur davon gesprochen, daß eine Aufgabe auf Schedulingebene durch sukzessive Ausführung von Basisfunktionen erfüllt werden kann. *Wie* dies geschieht ist noch unbekannt. In diesem Kapitel wird die verwendete Planungsstrategie beschrieben und erläutert.

4.1 Das Roboterprogramm

4.1.1 Der Programmspeicher

Der Roboter verfügt über einen linearen Programmspeicher. In ihm ist eine Sequenz von Befehlen abgelegt. Dies können einfache Basisfunktionen sein oder direkt vom Scheduler erhaltene Aufgaben. Auch die auf Seite 10 beschriebene Anweisung, eine benannte Ecke anzufahren, kann im Programm auftreten.

4.1.2 Programmausführung

Ist der Roboter bereit und sind noch nicht alle Programmschritte abgearbeitet, so wird der nächste Befehl ermittelt und ausgeführt. Drei verschiedene Situationen können auftreten:

1. Der nächste Befehl ist eine direkt ausführbare Aktion. In diesem Fall wird er aus dem Programm entfernt vom Roboter ausgeführt.

2. Der nächste Befehl ist eine Aufgabe auf Schedulingebene. Da für eine solche Aufgabe immer dieselben Aktionen nötig werden (siehe Abschnitt 3.2.1), genügt es, diese Aktionen für jede Aufgabe direkt abzuspeichern. Die Anwendung eines Planungsalorithmus wie etwa in [Bratko, 1990, Kap.16] beschrieben ist nicht erforderlich.

   Die Aufgabe wird aus dem Programm entfernt und durch die für sie notwendigen Unteraufgaben ersetzt. Die Programmausführung wird bei der ersten dieser Unteraufgaben fortgesetzt.

3. Die Anweisung, zu einer Ecke zu fahren, löst den Bewegungsplaner aus. Dieser liefert die notwendigen Bewegungen, um die Ecke vom derzeitigen Roboterstandort aus zu erreichen. Wie bei den Aufgaben werden diese Aktionen am Programmanfang eingefügt, nachdem die ursprüngliche Anweisung entfernt wurde.

Durch dieses Verfahren wird erreicht, daß nur dann eine Planung stattfindet, wenn sie wirklich erforderlich ist. Da sich die Bewegungsplanung immer auf den aktuellen Standort des Roboters bezieht, darf sie sogar zu keinem früheren Zeitpunkt durchgeführt werden.

4.1.3 Realisierung der Bewegungsplanung

Die für das Erreichen einer Ecke notwendigen Bewegungen werden in zwei Stufen ermittelt.

Kürzester Weg mit heuristischer Suche

Als erster Schritt wird der kürzeste Weg vom aktuellen Standort des Roboters zur Zielecke bestimmt. Dies geschieht mittels einer an [Bratko, 1990, Kap.12] angelehnten Best-first search.

Der Prolog-Code wurde dahingehend modifiziert, daß er ohne Veränderung auf verschiedene Probleme angesetzt werden kann. Dies ist durch Parametrisierung des Algorithmus mit einer Struktur, die die Teile enthält, die bei unterschiedlichen Problemen variieren. Dies sind die Erzeugung von Nachfolgern eines gegebenen Knotens im Suchbaum, das Kriterium für einen Zielknoten, die heuristische Schätzung der Entfernung eines Knotens vom Ziel und ein Wert, der verbindlich größer ist als alle jemals auftretenden Kosten.

Eine zweite Modifikation besteht darin, daß der Algorithmus neben der Lösung auch ihre Kosten liefert.

Für das Problem des kürzesten Weges von der Position des Roboters zu einer beliebigen Zielecke wurden die Parameter gewählt wie folgt:

1. Ein Knoten im Suchbaum wird durch ein Wertetripel repräsentiert, das eine Ecke, einen Winkel und eine Entfernung enthält. Die Ecke ist eine Position im Arbeitsbereich, der Winkel bezeichnet die Blickrichtung des Roboters bei Betreten dieser Ecke, und die Entfernung ist die Länge der Kante, die der Roboter als letztes entlanggefahren ist.

   Obwohl für eine simple Bestimmung des kürzesten Weges nur die Ecke relevant ist, wurde diese Darstellung gewählt, da sie bei der später folgenden Konstruktion des Bewegungsplanes dienlich ist.

2. Der Startknoten für die Suche ist gegeben durch die anfängliche Position des Roboters, seinen derzeitigen Blickwinkel und dem Wert Null.

3. Die Nachfolger eines Knotens sind alle Nachbarn der Ecke im Arbeitsbereich, verbunden mit dem jeweiligen Winkel und der Länge der verbindenden Kante.

4. Zielknoten sind diejenigen Knoten, deren Ecke mit der Zielecke identisch ist.

5. Als Schätzwert für einen Knoten gilt die Länge der direkten Luftlinie zwischen seiner Ecke und der Zielecke. Der heuristischen Wert dieser Schätzung kann so gesehen werden, daß eine Ecke im Arbeitsbereich, die näher am Ziel liegt, i.Allg. mit einer größeren Wahrscheinlichkeit auch Teil des kürzesten Weges dorthin ist.

   Diese Bewertungsfunktion ist *zulässig* (engl. admissible, vgl.[Bratko, 1990, Seite 286]), da der kürzeste Weg von einer Ecke zur anderen nur im günstigsten Fall der direkten Entfernung beider Ecken entspricht, nämlich dann, wenn dieser Pfad geradlinig verläuft. In allen anderen Fällen stellt der Weg durch den Graphen einen Umweg dar. Dies gilt nur, wenn der Graph wie der Arbeitsbereich in einer Ebene liegt.

   Durch die Zulässigkeit der Bewertungsfunktion wird dieser Suchalgorithmus zu einem A*-Algorithmus. Dieser liefert immer ein optimales Ergebnis, in diesem Fall den kürzesten Weg.

Die heuristische Suche wurde anderen Algorithmen wie z.B. Floyd-Warshall vorgezogen, da sie ein weitaus besseres Laufzeitverhalten zeigt. Der Floyd-Warshall-Algorithmus wäre nur anwendbar, wenn mit ihm sämtliche kürzesten Wege im Arbeitsbereich bei der Initialisierung berechnet und gespeichert würden. Die würde bei Arbeitsbereichen mit vielen Ecken eine untragbar lange Initialisierungsphase hervorrufen, da sich die Komplexität des Algorithmus kubisch zur Anzahl der Ecken verhält[1].

[1] siehe [Maas, 1993]

Der Dijkstra-Algorithmus berechnet die kürzesten Wege von einer vorgegebenen Ecke zu allen anderen Ecken eines Graphen. Deshalb eignet er sich weder zur Berechnung der kürzesten Wege zwischen allen Ecken während der Initialisierung, noch zur Ermittlung des kürzesten Weges zwischen nur zwei Ecken während der Simulation.

Der verwendete A*-Algorithmus berechnet den kürzesten Weg zwischen zwei Ecken schnell genug, sodaß er während der Simulation verwendet werden kann. Sollten Arbeitsbereiche mit vielen Ecken die Performanz zu sehr beeinträchtigen, kann der Algorithmus so modifiziert werden, daß einmal berechnete Wege gespeichert werden, um mehrfache Berechnung desselben Weges zu unterbinden.

Entwurf eines Bewegungsplans aus dem Ergebnis der Heuristischen Suche

Aus der bereits beschriebenen Darstellung des kürzesten Weges als Ausgabe des Suchalgorithmus kann nun ein Bewegungsplan erstellt werden, der diesen Weg entlangfährt. Es gibt jeweils drei Möglichkeiten, von einem Knoten des Pfades zum nächsten zu gelangen[2]. Sie sind hier in der Reihenfolge ihrer Priorität aufgeführt. Die weiter hinten in der Liste auftretenden Möglichkeiten werden nur in Erwägung gezogen, wenn keine der davor stehenden anwendbar ist.

1. Sind die Winkel dieses und des nächsten Knotens gleich, so kann der Roboter vorwärts fahren. Die Planung wird mit dem nächsten Knoten fortgesetzt.

2. Sind die Winkel der beiden ersten Knoten um 180° verschieden, wird der Roboter eine Rückwärtsbewegung durchführen. Wiederum wird die Planung mit dem Folgeknoten fortgesetzt.

3. Kann der nächste Knoten weder durch eine Vorwärts- noch eine Rückwärtsbewegung erreicht werden, so muß sich der Roboter drehen.

   Es wird immer die Drehung gewählt, die die Blickrichtung des Roboters mit dem Winkel des folgenden Knotens zur Deckung bringt. Dies bewirkt, daß der Roboter höchstens solange rückwärts fährt, bis er die erste Drehung vollzogen hat. Ab dann wird nur noch vorwärts gefahren.

   Die Planung wird mit dem aktuellen(!) Knoten fortgesetzt. Dessen Winkel wird jedoch durch den Blickwinkel des Roboters nach der vollzogenen Drehung ersetzt. Dies führt im nächsten Schritt zu eine Vorwärtsbewegung.

[2]Zur Erinnerung: Ein Knoten ist ein Tripel aus einer Ecke, einem Winkel und einer Entfernung.

Diese Schritte werden solange angewendet, bis nur noch der Zielknoten übrig ist. Es sei bemerkt, daß der resultierende Bewegungsplan nicht die optimale Lösung ist, da nach der ersten Drehung keine Rückwärtsbewegungen mehr in Erwägung gezogen werden. Dieser Effekt ist beabsichtigt, da sich der Roboter vorzugsweise vorwärts bewegen soll.

Kapitel 5

Fallbeispiel: Test eines Schedulingalgorithmus

5.1 Die Beispielumgebung

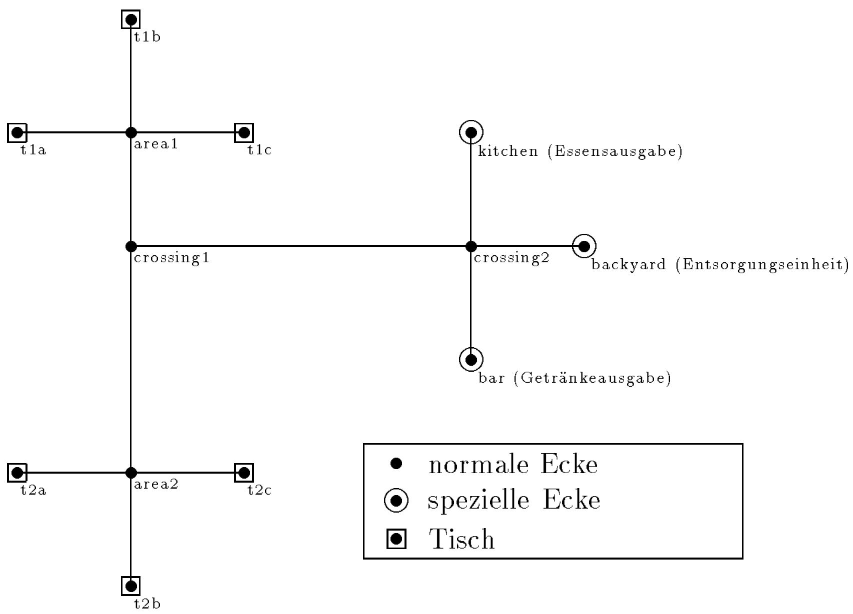

Abbildung 5.1: Karte des Beispielarbeitsbereiches

Der Arbeitsbereich, der für diese Fallstudie verwendet wurde, ist in Abbildung 5.1 skizziert. Er besteht aus insgesamt 13 Ecken, davon sind sechs Tische (*t1a,* *t1b*, *t1c*, *t2a*, *t2b* und *t2c*). Drei weitere Ecken sind jeweils als Essensausgabe (*kitchen*), Getränkeausgabe (*bar*) und Entsorgungseinheit (*backyard*) deklariert. Alles weitere sind einfache Ecken.

Die Anordnung der Ecken ist so gewählt, daß es zwei Gruppen zu je drei Tischen gibt. Alle Tische in einer Gruppe haben denselben Abstand zur Entsorgungseinheit, zur Essens- und zur Getränkeausgabe. Dieser Abstand ist für die eine Gruppe genau eine Längeneinheit kürzer als für die Andere.

Dies kann beispielsweise dazu dienen, zu beurteilen, ob ein unterschiedlicher Abstand eine Bevorzugung einer der Gruppen bei der Bedienung hervorruft.

5.2 Scheduling mit FCFS

Als Scheduling-Strategie wurde ein einfacher First-come-first-serve-Algorithmus gewählt. Neue Aufgaben werden hinten an eine Liste angehängt. Als nächste Aufgabe gilt immer das erste Element dieser Liste.

Es ist zu erwarten, daß diese Strategie eine ungünstige Reihenfolge in der Abarbeitung der Aufgaben bewirkt. Dies resultiert daraus, daß bei dem FCFS-Algorithmus keine Charakteristika der Aufgaben berücksichtigt werden.

Ein Vorteil dieses Verfahrens liegt in der einfachen Implementierung. Außerdem besteht keine Gefahr, daß ein Gast vernachlässigt wird, weil er an einem ungünstig gelegenen Tisch sitzt.

5.3 Durchgeführte Untersuchungen

Während der Simulationsläufe wurden grundsätzlich die voreingestellten Werte für die Systemparameter verwendet (siehe Abschnitt 3.3).

5.3.1 Grundlagen

Bedienrate des Systems

Die Information, welche Anzahl von Gästen vom System pro Stunde bedient werden können, kann als Kriterium dienen, verschiedene Schedulingstrategien miteinander zu vergleichen. Dieser Wert wird Bedienrate genannt. Es wird exemplarisch am FCFS-Algorithmus dargestellt, wie die Bedienrate des Systems ermittelt werden kann. Die Bedienrate wird von folgenden Faktoren beeinflußt:

- Der verwendeten Schedulingstrategie,
- dem Aufbau des Arbeitsbereiches,
- der Fahr- und Drehgeschwindigkeit des Roboters,
- den Herstellzeiten für Speisen und Getränke,
- dem Verhalten der Gäste.

Um also eine Aussage darüber treffen zu können, wie verschiedene Schedulingstrategien die Bedienrate beeinflussen, darf in den verschiedenen Tests nur der verwendete Algorithmus ausgetauscht werden. Alle anderen Systemparameter dürfen nicht variieren.

Damit der ermittelte Wert einer Approximation der maximalen Bedienrate entspricht, wurde das System während des Tests voll ausgelastet. Dies wurde erreicht, indem zum Systemstart eine sehr hohe Anzahl Gäste in die Warteschlange für die Tische eingefügt wurde. Solange sich Gäste in der Warteschlange befinden, wird jeder freiwerdende Tisch sofort wieder besetzt. Es wird somit die volle Kapazität des Systems ausgenutzt.

Es wird zu jeder vollen Stunde i (Simulationszeit) die Anzahl a_i der Gäste beobachtet, die das System bedient verlassen haben. Für $i = 1, ..., n$ und $a_0 = 0$ ist

$$x_i = ai - ai - 1$$

die Anzahl der in Stunde i bedienten Gäste. Wird diese Größe über einen Zeitraum von n Stunden betrachtet, so ergibt dies n Realisierungen der Zufallsgröße $\underline{x}$. Ihr arithmetisches Mittel

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

stellt eine Schätzung der durchschnittlich in einer Stunde bedienbaren Gästen dar. Die Genauigkeit dieser Schätzung kann mittels Konfidenzintervallen angegeben werden. Man spricht in diesem Fall von Konfidenzschätzungen.

Konfidenzschätzungen

Seien $x_i, i = 1, ..., n$ die Realisierungen einer Zufallsgröße $\underline{x}$, so stellt das arithmetische Mittel $\overline{x}$ aus Formel (5.3.1) einen Schätzwert für den tatsächlichen Erwartungswert $E[\underline{x}]$ der Grundgesamtheit dar. Für eine Konfidenzschätzung wird noch die empirische Streuung benötigt, die wie folgt definiert ist:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x}).$$

Um die Qualität dieser Schätzung zu ermitteln, wird die in [Siegert, 1991] hergeleitete Formel für die Fehlerschranke Θ

$$\Theta = \frac{k * s_n}{\sqrt{n}}$$

verwendet. Die Größe k steuert die Wahrscheinlichkeit ϵ, mit der $\overline{x}$ im Intervall $[E[\underline{x}] - \Theta, E[\underline{x}] + \Theta]$ liegt. Es besteht folgende Beziehung:

$$2 * \Phi(k) - 1 = \epsilon$$

wobei Φ die Verteilungsfunktion der standardisierten Normalverteilung ist. Für häufig verwendete Werte von ϵ gibt es Tabellen, aus denen k bestimmt werden kann[1]. Für größere Werte von ϵ, also eine höhere Sicherheit der Aussage, gilt, daß deren Genauigkeit abnimmt, da sich die Fehlerschranke Θ und damit das Konfidenzintervall vergrößert.

Um gleichermaßen die Sicherheit und die Genauigkeit einer Schätzung zu erhöhen, muß die dafür verwendete Stichprobe einen möglichst großen Umfang n besitzen. Wie aus (5.3.1) erkennbar ist, verhält sich die Fehlerschranke Θ umgekehrt proportional zur Quadratwurzel aus n. Dies bedeutet, daß eine Stichprobe vom vierfachen Umfang notwendig ist, um die Fehlerschranke bei gleicher Konfidenz zu halbieren.

5.3.2 Geschätzte Bedienrate des Systems

Für den durchgeführten Test wurde der in 5.1 beschriebene Arbeitsbereich verwendet. Die Warteschlange für die Tische wurde mit 10000 Gästen initialisiert. Während der Simulation wurde die Ankunft weiterer Gäste unterbunden. Beobachtet wurde die Anzahl der bedienten Gäste zu jeder vollen Stunde Simulationszeit.

Über (5.3.1) wurde errechnet, wieviel Gäste jeweils in einer Stunde bedient wurden. Das arithmetische Mittel und die empirische Streuung dieser Stichprobe ergaben sich zu

[1] z.B. in [Storm, 1988] auf Seite 143, oder [Siegert, 1991] auf Seite 168.

$$\overline{x} = 6.72$$

und

$$s^2 = 1.127297.$$

Das arithmetische Mittel stellt eine Schätzung der tatsächlichen Anzahl der in einer Stunde bedienbaren Gäste, also der Bedienrate, dar. Mit dem in Abschnitt 5.3.1 beschriebenen Verfahren wurde die Fehlerschranke für ein 95%-Vertrauensintervall ermittelt:

$$\Theta = \frac{k * s_n}{\sqrt{n}} = \frac{1.96 * 1.127297}{\sqrt{10000}} = 0.069367$$

Die tatsächliche Bedienrate liegt also mit einer Sicherheit von 95% in dem Intervall $[6.72 - 0.069367; 6.72 + 0.069367]$, also zwischen 6.65 und 6.79.

Kapitel 6

Fazit und weiterführende Themen

Es wurde ein System entworfen, in dem einem Roboter zufällig parallel verschiedene Aufgaben gestellt werden. Es bietet die Möglichkeit, verschiedene Schedulingstrategien zur Planung der Ausführungsreihenfolge dieser Aufgaben einzubinden und anhand von Statistiken miteinander zu vergleichen.

Während der Testläufe wurde das System in Bezug auf schnelle Durchführung der Simulation verbessert. So wurde zum Beispiel durch Implementierung einiger kritischer Funktionen in C/C++ eine deutlich höhere Performanz erreicht.

Der Mechanismus zur Beobachtung der statistischen Werte erwies sich als unzureichend in Bezug auf das Hinzufügen neuer Werte und allgemeine Übersichtlichkeit. Die Ausgabe geschieht teilweise dezentral in verschiedenen Modulen. Die in [Siegert, 1991] beschriebenen Modellbereiche könnten eine Lösung dieses Problems bieten.

Das Simulationsmodell und dessen Implementierung ist teilweise komplexer als erforderlich. Dies liegt an der Tatsache, daß der Schwerpunkt der Studienarbeit erst in ihrem Verlauf in den Bereich der Simulation gerückt ist.

Die Anwendung des Systems steht noch aus. Verschiedene Schedulingalgorithmen müssen entworfen und implementiert werden. Für deren Bewertung und Vergleich müssen Kriterien zusammengestellt und angewendet werden.

Auch eine bessere Visualisierung der simulierten Vorgänge stellt eine interessante Aufgabe dar. Eine grafische Darstellung des Arbeitsbereiches, des Roboters, der Gäste und der Objekte wäre weitaus attraktiver als die derzeit ausgegebenen Textmeldungen.

Als großes Ziel besteht natürlich die Anbindung an einen wirklichen Roboter.

Anhang A

Dokumentation der einzelnen Module

Dieser Anhang enthält die Dokumentation der einzelnen Programmteile. Für jedes Modul werden die enthaltenen Prädikate aufgeführt und erklärt. Hilfsprädikate, die nur zur Implementation der eigentlichen Prädikate dienen, werden nicht gesondert erwähnt.

A.1 Basisfunktionalität

A.1.1 Strukturierter Speicher: *fetch.pl*

Um den Zustand der Komponenten während der Simulation speichern zu können, muß eine Möglichkeit zur Datenspeicherung vorhanden sein. Die Prolog-Datenbank mit dem assert/retract-Mechanismus erwies sich für diesen Zweck als zu langsam. Daher wurden folgende Prädikate eingeführt.

store(*+Key1, +Key2, +Value*)
: Der Datenwert *Value* wird unter dem aus *Key1* und *Key2* gebildeten Schlüsseltupel abgelegt.

fetch(*+Key1, +Key2, ?Value*)
: Ist ein Wert mit dem Schlüssel *(Key1, Key2)* assoziiert, wird dieser mit *Value* unifiziert.

free(*+Key1, +Key2*)
: Sind *Key1* und *Key2* mit einem Wert assoziiert, löst dieses Prädikat die Assoziation und gibt die verwendeten Resourcen frei. Ansonsten schlägt es fehl.

free(*+Key1*)
: Alle vorhandenen Assoziationen, die *Key1* als erstes Schlüsselattribut besitzen, werden aufgelöst.

Diese Schnittstelle entkoppelt die Implementation des Speichermechanismus von den übrigen Modulen. Um ein gutes Laufzeitverhalten zu erreichen, wurde eine Shared Library in C/C++ implementiert, die über das in [Wielemaker, 1996] beschriebene Foreign-Language-Interface von SWI-Prolog angesprochen wird.

Für Systeme, die diesen Mechanismus nicht unterstützen, ist eine Prolog-Implementation von `store/3`, `fetch/3`, `free/[1,2]` in der Datei *pl-fetch.pl* definiert. Diese wird automatisch geladen, falls die Shared Library *libfetch.so* nicht geöffnet werden kann. In diesem Fall muß jedoch mit einem deutlich schlechteren Laufzeitverhalten gerechnet werden.

A.1.2 Der Nachrichtenmechanismus: *mail-bag.pl*

registerReceiver(*+Receiver, +Handler*)
: Jede Systemkomponente, die Nachrichten empfangen soll, muß über dieses Prädikat registriert werden. *Receiver* muß ein eindeutiger Name sein, der dann beim Versenden von Nachrichten als Empfängerangabe verwendet werden kann.

  Handler ist der Name eines einstelligen Prädikats. Beim Zustellen von Nachrichten wird der Handler jedes registrierten Empfängers mit dessen Namen als Argument aufgerufen. Über `receiveMessage/3` können dann die Nachrichten empfangen und verarbeitet werden.

unregisterReceiver(*+Receiver*)
: Die Registrierung des Empfängers *Receiver* wird aufgehoben. Er kann nun keine Nachrichten mehr empfangen.

deliverMessages
: Alle Empfänger mit ungelesenen Nachrichten werden durch Aufruf ihres Handlers zum Lesen ihrer Nachrichten veranlaßt.

queueDelivering(*+Time*)
: Der Aufruf dieses Prädikats stellt sicher, daß in spätestens der durch *Time* angegebenen Zeitspanne `deliverMessages/0` aufgerufen wird. Weitere Aufrufe werden ignoriert, bis eine Nachrichtenzustellung stattgefunden hat.

sendMessage(*+Sender, +Receiver, +Message*)
Es wird die Nachricht *Message* von *Sender* an *Receiver* gesendet. *Receiver* muß als Empfänger registriert sein. *Message* darf ein beliebiger Prolog-Ausdruck sein, dessen Verarbeitung vom Empfänger durchgeführt werden muß.

receiveMessage(*+Receiver, ?Sender, ?Message*)
Dieses Prädikat dient dazu, anstehende Nachrichten zu lesen. Es ist zur Verwendung in Nachrichtenhandlern vorgesehen.

Der folgende Quelltextauschnitt verdeutlicht den Aufbau eines Nachrichtenhandlers.

```
messageHandler(Self) :-
  receiveMessage(Self, Sender, Message),
  handleMessage(Self, Sender, Message),
  fail.

messageHandler(_).

handleMessage(Self, Sender, Message) :-
  writef('%w hat folgende Nachricht von %w empfangen: %w',
         [Self, Sender, Message]), !.
```

A.1.3 ADT für Warteschlangen: *queue.pl*

Diese Datei stellt die Funktionalität von Warteschlangen bereit, die nach dem FIFO-Prinzip arbeiten.

initQueue(*+QueueName*)
Es wird eine neue, leere Warteschlange mit dem Namen *QueueName* angelegt.

initQueue(*+QueueName, +ElemList*)
Es wird eine neue Warteschlange mit dem Namen *QueueName* angelegt und mit den Elementen der Liste *ElemList* initialisiert. Das erste Element von *ElemList* ist auch das erste Element in der Warteschlange.

enqueue(*+QueueName, +Element*)
Das Element *Element* wird in die Warteschlange *QueueName* eingefügt.

dequeue(*+QueueName, ?Element*)
Läßt sich das erste Element in der Warteschlange *QueueName* mit *Element* unifizieren, wird dieses aus der Warteschlange entfernt. Ansonsten schlägt die Operation fehl.

queueFront(*+QueueName, ?Element*)
Das erste Element in der Warteschlange *QueueName* wird mit *Element* unifiziert, ohne entfernt zu werden.

emptyQueue(*+QueueName*)
Dieses Prädikat ist wahr, wenn die Warteschlange *QueueName* leer ist.

queueLength(*+QueueName, ?Length*)
Length wird mit der Länge der Warteschlange *QueueName* unifiziert.

A.1.4 Verschiedenes: *misc.pl*

initContainer(*+Container*)
Dieses Prädikat initialisiert eine Datenstruktur mit dem Namen *Container*. Es können Objekte in diese Datenstruktur eigefügt und aus ihr entfernt werden. Jeder Knoten des Arbeitsbereiches und der Roboter verwenden diesen Mechanismus.

hasObject(*+Container, +Object*)
Prüft, ob *Container* das Objekt *Object* enthält.

addObject(*+Container, +Object*)
Fügt das Objekt *Object* dem Inhalt von *Container* hinzu.

removeObject(*+Container, +Object*)
Entfernt das Objekt *Object* aus dem Inhalt von *Container*.

announcement(*+Format, +Arguments, +AnnouncementType*)
Die Prädikate `announcement/[2,3]` werden für alle Meldungen des Systems während der Simulation verwendet.

Die Ausgabe entspricht dem des Aufrufs `writef(Format, Arguments)`. Es wird sichergestellt, daß die aktuelle Simulationszeit aus der Ausgabe ersichtlich ist.

AnnouncementType	**Bedeutung**
`log`	allgemeine Ausgaben
`message-reading`	Lesen von Nachrichten
`message-receiving`	Empfangen von Nachrichten
`message-sending`	Senden von Nachrichten
`guests`	Informationen über Gäste
`robot_tasks`	Neue Aufgaben für den Roboter
`robot_state`	Statusmeldungen des Roboters
`scheduler`	Meldungen des Schedulers
`error`	Fehlermeldungen

Tabelle A.1:

Das Argument *AnnouncementType* bestimmt die Herkunft der Ausgabe. Es wird dazu verwendet, Ausgaben aus bestimmten Bereichen zu erlauben oder zu unterdrücken. Tabelle A.1 enthält die möglichen Werte für *AnnouncementType* und ihre Bedeutung.

Um die Ausgabe von Meldungen zu erlauben oder zu unterdrücken, werden die Prädikate `doAnnounce/1` und `doNotAnnounce/1` verwendet.

announcement(*+Format, +AnnouncementType*)
Entspricht `announcement(Format, [], AnnouncementType)`.

doAnnounce(*+Type*)
Ausgaben vom Typ *Type* werden erlaubt. Näheres siehe `announce/3`.

doNotAnnounce(*+Type*)
Ausgaben vom Typ *Type* werden unterdrückt. Näheres siehe `announce/3`.

sumList(*+List, -Sum*)
Ist *List* eine Liste von Zahlen, so wird *Sum* deren Summe zugewiesen.

invertList(*+List, -Inversion*)
Kehrt die Liste *List* um. *Inversion* enthält das Ergebnis.

exponential(*?Y, +Alpha, ?X*)
Berechnet für gegebenes X und *Alpha* den Wert der Verteilungsfunktion der Exponentialverteilung mir Parameter *Alpha* an der Stelle X. Ist statt X das Argument Y gebunden, so wird der Wert von X über die Umkehrfunktion der Verteilungsfunktion ermittelt.

randExp(*+Alpha, -X*)
Erzeugt Zufallszahlen X, die entsprechend der Exponentialverteilung mit dem Parameter *Alpha* verteilt sind. Dieses Prädikat wird verwendet, um Ereignisse in exponential verteilten Zeitabständen auftreten zu lassen.

constantly(*?X, ?Y*)
Die Argumente X und Y werden unifiziert. Dieses Prädikat dient zur Erzeugung von Ereignissen in konstanten Zeitabständen.

randomElement(*+List, -Element*)
Es wird ein beliebiges Element *Element* aus der Liste *List* ausgewählt.

turn(*+Angle, ?TurnAngle, ?NewAngle*)
Angle, *TurnAngle* und *NewAngle* sind Winkelangaben im Gradmaß zwischen 0° und 360°. *NewAngle* stellt das Ergebnis der Drehung um *TurnAngle* dar, ausgehend von *Angle*. *TurningAngle* kann auch negative Werte annehmen, was eine Drehung entgegen dem Uhrzeigersinn bedeutet.

distance(*+P1, +P2, -Distance*)
P1 und *P2* stellen geometrische Positionsangaben der Form *(X, Y)* dar. Berechnet wird die Enfernung *Distance* zwischen beiden Punkten.

angle(*+P1, +P2, -Angle*)
P1 und *P2* stellen geometrische Positionsangaben der Form *(X, Y)* dar. Es wird die Blickrichtung von *P1* nach *P2* in Grad berechnet.

A.2 Der Simulationskern

A.2.1 Laden und Initialisierung: *loader.pl*

init(*+Scheduler, +Workspace, +Timescale, +MaxGuests, +GuestFrequency, +InitialGuests, +StartNode, +StartAngle*)
Mit diesem Prädikat wird das Simulationssystem initialisiert. *Scheduler* und *Workspace* sind die Namen der Dateien, die die Implementation des Schedulingalgorithmus bzw. der Definition des Arbeitsbereiches enthalten.

Timescale gibt das Verhältnis der Simulationszeit zu realen Zeit an. Am häufigsten werden die Werte 0 (schnellstmögliche Simulation) und 1 (Simulation in Realzeit) verwendet.

MaxGuests gibt die Zahl der Gäste an, die insgesamt während der Simulation eintreten werden. *GuestFrequency* bestimmt die Anzahl der Gäste, die durchschnittlich in einer Stunde eintreten. *InitialGuests* Gäste werden vor Beginn der Simulation in die Warteschlange der Rezeption eingefügt. Die Gesamtzahl der Gäste ist somit die Summe von *InitialGuests* und *MaxGuests*.

StartNode und *StartAngle* bestimmen die Position und Blickrichtung des Roboters zu Beginn der Simulation.

run

Die Simulation wird laufen gelassen, bis keine neuen Ereignisse in der Ereignisliste stehen.

run(+*Time*)

Wie `run/0`, nur daß die Simulation spätestens nach der durch *Time* festgelegten Zeit abgebrochen wird. Die Zeit muß in Sekunden angegeben werden.

A.2.2 Die Ereignisliste: *timesim.pl*

initTimesim(+*Timescale*)

Initialisiert die Ereignisliste. *Timescale* gibt das Verhältnis der Simulationszeit zu realen Zeit an (siehe `init/8`).

timesim

Dieses Prädikat stellt die Hauptschleife der Simulation dar. Es wird von `run/[0,1]` aufgerufen.

stopSimulation

Nach Aufruf dieses Prädikats wird die Simulation zum nächstmöglichen Zeitpunkt gestoppt.

queueEvent(+*Event*, +*Timespec*)

Das Ereignis *Event* wird entsprechend der in *Timespec* spezifizierten Zeitangabe in die Ereignisliste eingetragen. *Timespec* kann folgende Formen haben:

emptyAgenda

Mit diesem Prädikat kann die Ereignisliste vollständig geleert werden. Die geschieht normalerweise bei der Systeminitialisierung.

AbsTime	Ein absoluter Zahlenwert (in Sekunden). Dieser muß gleich oder größer als die aktuelle Simulationszeit sein.
RelTime `from now`	Eine Zeitangabe, die relativ zur aktuellen Simulationszeit *RelTime* Sekunden in der Zukunft liegt. Der Ausdruck `0 from now` ergibt die aktuelle Simulationszeit.
`frequently(`*Freq*`)`	Das Ereignis trifft regelmäßig ein. Der Abstand *Interval* zum jeweils folgenden Ereignis wird durch den Aufruf von `apply(Freq, [Interval])` ermittelt. *Freq* kann auch das Format *Freq* `*` *N* haben. So kann über *N* angegeben werden, wie oft das Ereignis eintreffen soll. Für *Freq* können z.B. die Prädikate `randExp/2` und `constantly/2` verwendet werden.
`frequently(`*Freq*`, unimportant)`	Wie `frequently(Freq)`, das dazugehörige Ereignis wird jedoch als *unwichtig* eingestuft. Die Simulation stoppt, wenn sich nur noch unwichtige Ereignisse in der Ereignisliste befinden. Dies kann dazu verwendet werden, um etwa ein Ereignis zu erzeugen, das regelmäßig Statistikwerte in eine Datei schreibt.

Tabelle A.2: Zeitspezifikationen für Ereignisse

A.2.3 Statistikerfassung: *statistics.pl*

initStatistics

Hier werden die notendigen Initialisierungen für die Statistikerfassung vorgenommen.

logStatistics(*+StatisticList, +File, +Seconds*)

Mit diesem Prädikat wird die regelmäßige Ausgabe der in *StatisticList* angegebenen Statistikwerte in die durch *File* spezifizierte Datei veranlaßt. Das Argument *Seconds* bestimmt die Simulationszeit in Sekunden, die zwischen zwei Ausgaben liegen soll. *StatisticList* ist eine beliebige Liste der in Tabelle A.3 beschriebenen Werte.

Wert	**Bedeutung**
`elapsed_time`	bisher simulierte Zeit in Sekunden
`arrived_guests`	bisher in das System eingetretene Gäste
`served_guests`	Anzahl der Gäste, die das System bedient verlassen haben
`average_arrival_time`	durchschnittlicher Zeitraum zwischen zwei eintreffenden Gästen
`distance_per_guest(`*+Robot*`)`	Die Mittlere Entfernung, die vom Roboter *Robot* für die Bedienung eines Gastes zurückgelegt wurde

Tabelle A.3: Verfügbare Statistikwerte

verboseStatistics

Erzeugt die Ausgabe aller verfügbaren Statistikwerte.

incrementArrivedGuests(*+X*)

Der Zähler für die eingetretenen Gäste wird um den Wert X erhöht.

incrementServedGuests

Der Zähler für die bedienten Gäste wird um Eins inkrementiert. Außerdem wird die Zeit ermittelt, die seit dem letzten Austritt eines Gastes aus dem System vergangen ist, und in der Datei *served_interval.log* fortgeschrieben.

A.3 Der Roboter

A.3.1 Die Roboterschnittstelle: *robot.pl*

initRobot(*+Robot, +Node, +Angle*)
Der Roboter *Robot* wird initialisiert. Seine Startposition und Blickrichtung werden mit den Parametern *Node* und *Angle* angegeben.

coveredDistance(*+Robot, -Distance*)
Während der Simulation wird die von einem Roboter zurückgelegte Entfernung gespeichert. Auf diese Information kann über dieses Prädikat zugegriffen werden.

Die in Tabelle A.4 enthaltenen Prädikate lösen bei Robotern die in Abschnitt 3.2.1 beschriebenen Handlungen aus. Sie werden niemals direkt aufgerufen, sondern nur bei laufender Simulation über das Roboterprogramm[1]. Damit dieser Mechanismus funktioniert, müssen alle diese Prädikate als erstes Argument den Namen des ausführenden Roboters erwarten.

Prädikat	Ausgelöste Handlung
goForward(*+Robot*)	Vorwärts fahren
goBackward(*+Robot*)	Rückwärts fahren
turnLeft(*+Robot, +Angle*)	Links drehen
turnRight(*+Robot, +Angle*)	Rechts drehen
getOrders(*+Robot*)	Bestellung entgegennehmen
getObject(*+Robot, +Object*)	Objekt nehmen
putObject(*+Robot, +Object*)	Objekt abstellen
cashGuest(*+Robot, +Guest*)	Kassieren
notify(*+Robot, +Receiver, +Message*)	
taskFinished(*+Robot, +TaskID*)	

Tabelle A.4: Zuordnung von Prädikaten zu Handlungen

Eine Ausnahme sind `notify/3` und `taskFinished/2`. Diese Prädikate sind existieren aus Implementationsgründen.

notify(*+Robot, +Receiver, +Message*)
Es wird die Nachricht *Message* vom Roboter *Robot* an den Empfänger *Receiver* gesendet.

[1] siehe Anhang A.3.4

taskFinished(*+Robot, +TaskID*)

Der Roboter *Robot* hat die Aufgabe *TaskID* beendet. Er wird auf das Empfangen neuer Aufgaben vorbereitet. Der Scheduler wird von der Erfüllung der Aufgabe benachrichtigt.

A.3.2 Heuristische Suche: *best1st.pl*

bestfirst(*+BestFirstParam, +Start, -Solution, -Cost*)

Best-First-Suche in dem durch *BestFirstParam* und *Start* beschriebenen Problemraum.

BestFirstParam ist ein Quadrupel der Form `(MaxF, PredSuccessor, PredHeuristic, PredGoal)`. Tabelle A.5 enthält die Beschreibung der einzelnen Elemente.

Das folgende Beispiel soll die Verwendung von `bestfirst/4` verdeutlichen. Es berechnet mit seiner Hilfe den kürzesten Weg in einem geometrischen Graphen.

```
%
% Berechnung des kuerzesten Weges von A nach B in einem
% geometrischen Graphen. Die Kanten des Graphen sind
% zusammen mit ihrer Laenge in edge/3 gespeichert.
% Das Praedikat airLineDistance/3 berechnet die direkte
% Entfernung zwischen zwei beliebigen Ecken.
%
shortestPath(A, B, Path, Cost) :-
  bestfirst(
    (
      99999,                                      % MaxF
      (Node, Succ, Cost, edge(Node, Succ, Cost)),   % PredSuccessor
      (Node, HVal, airLineDistance(Node, B, HVal)), % PredHeuristic
      (Node, Node = B)                            % PredGoal
    ),
    A, Path, Cost)
```

A.3.3 Bewegungsplanung: *motion-plan.pl*

motionPlan(*+Robot, +Dest, -Plan, -Cost*)

Ermittelt eine Folge von Drehungen und Bewegungen, die den Roboter *Robot* von seinem aktuellen Standort zu der durch *Dest* angegebenen Ecke des Arbeitsbereiches führen. Der berechnete Bewegungsplan *Plan* ist eine

Element	**Struktur**	**Erklärung**
MaxF	*atomar*	Ein Wert, der höher als jede mögliche Bewertung eines Knotens im Suchbaum ist.
PredSuccessor	(*Node, Succ, Cost, Pred*)	*Pred* wird während der Suche benutzt, um für einen gegebenen Knoten *Node* im Suchbaum über Backtracking alle Nachfolger *Succ* zu erzeugen. In *Cost* müssen die Kosten zurückgegeben werden, die ein Wechsel von *Node* nach *Succ* bedeutet. *Node*, *Succ* und *Cost* sind uninstantiierte Variablen, die innerhalb von *Pred* auftauchen und durch dessen Aufruf gebunden werden.
PredHeuristic	(*Node, HVal, Pred*)	*Pred* soll in *HVal* einen heuristischen Wert liefern, der die Kosten zwischen dem Knoten *Node* und dem Ziel der Suche schätzt.
PredGoal	(*Node, Pred*)	*Pred* soll für alle Knoten *Node* wahr sein, die die Zielbedingungen der Suche erfüllen.

Tabelle A.5: Elemente der Beschreibungsstruktur für die Best-First-Suche

Liste von Aktionen, deren Kosten *Cost* die für ihre Ausführung benötigte Zeit ist.

Der verwendete Algorithmus ist in Abschnitt 4.1.3 beschrieben.

A.3.4 Das Roboterprogramm: *program.pl*

Das Programm eines Roboters ist eine Liste von Handlungen, die von diesem sequentiell ausgeführt werden. Alle in Tabelle A.4 auf Seite 38 aufgeführten Prädikate können als Befehl in einem Roboterprogramm erscheinen. Das erste Argument dieser Prädikate ist immer der ausführende Roboter. Da ein Programm genau einem Roboter zuzuordnen ist, braucht diese Information nicht redundant in jedem Befehl gespeichert zu werden, sondern wird automatisch hinzugefügt.

Ein Beispiel: Der Roboter `r2d2` soll sich um 30° nach links drehen. Der direkte Aufruf hätte die Form `turnLeft(r2d2, 30)`, wogegen im Programm von `r2d2` `turnLeft(30)` stehen müßte.

Es gibt besondere Befehle, die nicht direkt ausführbar sind. Dies sind Aufgaben auf Schedulingebene und Anweisungen, zu einer Ecke zu fahren. Sie werden entsprechend dem in Abschnitt 4.1 beschriebenen Verfahren behandelt.

Eine Aufgabe auf Schedulingebene *Task* wird vom Scheduler mit einem eindeutigen Schlüssel *TaskID* assoziiert und in der Form `task(`*TaskID*, *Task*`)` in ein Roboterprogramm eingefügt.

Die Anweisung, zu der Ecke *Node* zu fahren, hat die Gestalt `goto(`*Node*`)`.

robotReplaceProgram(*+Robot, +Program*)
Das Programm des Roboters *Robot* wird durch die Liste der Befehle in *Program* ersetzt.

robotAppendProgram(*+Robot, +Program*)
Die Befehlsliste *Program* wird an das Programm von Roboter *Robot* angehängt.

robotNextAction(*+Robot*)
Der Roboter *Robot* führt den nächsten Befehl in seinem Programm aus. Ist sein Programm leer, so benachrichtigt er den Scheduler über seine Verfügbarkeit.

A.4 Andere Simulationskomponenten

A.4.1 Die Getränkeausgabe: *beverage.pl*

initBeverageProvider(*+Node*)
Hier werden die nötigen Initialisierungen für die Benutzung von der Ecke *Node* als Getränkeausgabe durchgeführt.

processBeverageVouchers(*+Provider*)
Alle auf der Ecke befindlichen Getränkebestellungen werden verarbeitet. Die Fertigstellung der bestellten Speisen wird in die Ereignisliste eingetragen.

beverageMenu(*-Menu*)
Menu wird mit einer Liste der Namen aller bestellbaren Getränke instantiiert.

A.4.2 Die Essensausgabe: *food.pl*

initFoodProvider(*+Node*)
Nach dem Aufruf dieses Prädikats kann die Ecke *Node* die Funktion der Essensausgabe übernehmen.

processFoodVouchers(*+Provider*)
Alle auf der Ecke befindlichen Essensbestellungen werden verarbeitet. Die Fertigstellung der bestellten Speisen wird in die Ereignisliste eingetragen.

foodMenu(*-Menu*)
Wie `beverageMenu/1`, nur daß hier eine Namensliste aller Speisen erzeugt wird.

A.4.3 Die Entsorgungseinheit: *disposal.pl*

initDisposalUnit(*+Node*)
Initialisiert die Ecke *Node* als Entsorgungseinheit.

disposeObjects(*+Node*)
Ist *Node* die Entsorgungseinheit, werden alle dort befindlichen Objekte aus dem System genommen.

A.4.4 Die Rezeption: *reception.pl*

initReception(*+InitialGuests*)
Die Rezeption wird initialisiert. Dies muß geschehen, bevor der erste Gast eintrifft. *InitialGuests* ist die Zahl der Gäste, die sich zu Beginn in der Warteschlange für die Tische befinden sollen.

A.4.5 Die Gäste: *guests.pl*

initGuests(*+GuestsPerHour, +MaxGuests*)
Führt alle Initialisierungen durch, die die Gäste betreffen. Während der Simulation werden insgesamt *MaxGuests* hintereinander eintreten.

GuestsPerHour gibt die mittlere Anzahl eintreffender Gäste pro Stunde Simulationszeit an.

guestPosition(*+GuestID, -Position*)
Unifiziert die Position, also den Tisch des durch *GuestID* eindeutig identifizierten Gastes mit dem Argument *Position*.

guestOrders(*+GuestID, -Orders*)
Sobald sich ein Gast an einen Tisch setzt, wird eine zufällige Liste von Getränken und Speisen erzeugt, die seine Bestellung darstellt. Die zu dem durch *GuestID* spezifizierten Gast gehörende Liste wird mit *Orders* unifiziert.

A.4.6 Die Tische: *tables.pl*

initTable(*+Node*)
Die Ecke *Node* wird als freier Tisch initialisiert.

freeTable(*?Node*)
Wahr für alle Tische, an denen sich kein Gast aufhält.

occupyTable(*+Table, +Guest*)
Wenn sich am Tisch *Table* kein Gast befindet, wird er vom Gast *Guest* besetzt.

leaveTable(*+Table, +Guest*)
Wenn sich der Gast *Guest* am Tisch *Table* aufhält, verläst der Gast den Tisch. Der Tisch ist wieder frei.

guestAtTable(*+Table, ?Guest*)

Ist der Tisch *Table* nicht besetzt, schlägt dieses Prädikat fehl. Ansonsten wird der Gast an diesem Tisch mit *Guest* unifiziert.

A.4.7 Der Arbeitsbereich: *workspace.pl*

initWorkspace

Hier werden aus den Pädikaten, die den Arbeitsbereich beschreiben, häufig benötigte Daten abgeleitet und gespeichert. Alle Elemente des Arbeitsbereiches werden initialisiert. Dies sind alle Ecken, Tische, die Entsorgungseinheit, die Essens- und die Getränkeausgabe.

Folgende Prädikate beschreiben einen Arbeitsbereich. Es sei darauf hingewiesen, daß eine Ecke gleichzeitig Entsorgungseinheit, Essens- und Getränkeausgabe sein kann.

node(*NodeName, (XPos, YPos)*)

Für jede Ecke des Arbeitsbereiches. *NodeName* muß ein Atom sein, das einen eindeutigen Namen für die jeweilige Ecke darstellt. *(XPos, YPos)* ist ihr aus beliebigen Zahlenwerten bestehende Ortsvektor.

link(*Node1, Node2*)

Ein Eintrag dieser Form bedeutet, daß eine Kante zwischen den durch *Node1* und *Node2* bezeichneten Ecken existiert. Diese Kante ist bidirektional, d.h. `link(a, b)` impliziert `link(b, a)`.

disposalUnit(*NodeName*)

Die Ecke *NodeName* wird zur Entsorgungseinheit erklärt. Für jeden Arbeitsbereich muß genau ein Eintrag dieser Form vorhanden sein.

foodProvider(*NodeName*)

Erklärt die Ecke *NodeName* zur Essensausgabe. Auch hier ist genau ein Eintrag für einen Arbeitsbereich notwendig.

beverageProvider(*NodeName*)

Wie `disposalUnit/1` und `foodProvider/`. Die Ecke *NodeName* wird zur Getränkeausgabe gemacht.

table(*NodeName*)

Die Ecke *NodeName* ist ein Tisch.

Die folgenden Informationen werden aus der Definition des Arbeitsbereiches durch `initWorkspace/0` abgeleitet und stehen zur Laufzeit zur Verfügung.

node2node(*?Node1, ?Node2, ?Dist, ?Angle*)
Für jedes Paar von Eckennamen *Node1* und *Node2* ist dies die Information, in welcher Richtung *Angle* und in welcher Entfernung *Dist* sich *Node2* relativ zu *Node1* befindet.

edge(*?Node1, ?Node2, ?Dist, ?Angle*)
Für alle verbundenen Eckenpaare enthält `edge/4` dieselben Informationen wie `node2node/4`.

adjacent(*?Node1, ?Node2*)
Dies ist eine vereinfachte Schreibweise für `edge(Node1, Node2, _, _)`.

A.5 Die Schnittstelle für Scheduler: *scheduler.pl*

Diese Datei enthält Prädikate, die eine Abstraktionsebene zwischen das Simulationssystem und die eigentliche Implementation des Schedulingalgorithmus einfügt. Der Zweck dieses Vorgehens ist, daß die Aspekte, die allen Implementationen gemeinsam sind, nur einmal kodiert werden müssen.

initSchedulerInternal
Hier werden alle Initialisierungen durchgeführt, die die Schnittstelle zwischen dem Scheduler und der Simulation betreffen. Die Initialisierung des eigenlichen Schedulingalgorithmus wird durch Aufruf von `initScheduler/0` ermöglicht.

schedulerAddTask(*-TaskID, +Task*)
Dieses Prädikat vergibt eine eindeutige Identifikation *TaskID* für die Aufgabe *Task*. *Task* muß ein in Tabelle A.4 auf Seite 38 enthaltenes Prädikat sein.

Die Aufgabe wird durch `addTask(TaskID, Task)` an den eigentlichen Schedulingalgorithmus übergeben.

schedulerAddTaskChain(*-TaskID, TaskList*)
TaskList ist eine Liste von Aufgaben, wie sie von `schedulerAddTask/2` erwartet werden. Die erste Aufgabe in der Liste wird an den Scheduler zur Einplanung übergeben. Die Erfüllung einer Aufgabe der Liste löst die Einplanung ihres Listennachfolgers aus. *TaskID* ist die Identifikation der ersten Aufgabe in der Liste.

schedulerPeekTask(-*TaskID*, -*Task*)
Ermittelt über den Aufruf `peekTask(TaskID, Task)` die nächste anstehende Aufgabe.

schedulerRemoveTask(+*TaskID*)
Die Aufgabe mit der Identifikation *TaskID* wird über `removeTask(TaskID)` aus der Menge der einzuplanenden Aufgaben entfernt.

Für jede neue Implementation eines Schedulingalgorithmus müssen folgende Prädikate implementiert werden. Das geforderte Verhalten ist ebenfalls beschrieben. Die Datei mit der jeweiligen Implementation muß vor Beginn der Simulation geladen werden.

initScheduler
Diese Funktion wird nur einmal zum Systemstart aufgerufen. In ihr sollte der Schedulingalgorithmus alle notwendigen Initialisierungen durchführen.

addTask(+*TaskID*, +*Task*)
Hiermit wird dem Algorithmus eine neue Aufgabe zur Einplanung übergeben. *Task* bezeichnet die Aufgabe, *TaskID* ist ein eindeutig identifizierender Bezeichner.

peekTask(-*TaskID*, -*Task*)
Diese Anfrage sollte die nächste anstehende Aufgabe und ihren eindeutigen Bezeichner liefern. Ist keine Aufgabe zu erledigen, muß die Anfrage fehlschlagen.

removeTask(+*TaskID*)
Mit dieser Anfrage wird der Scheduler dazu aufgefordert, die spezifizierte Aufgabe aus der Planung zu entfernen.

Literaturverzeichnis

[Bratko, 1990] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 2nd Edition, MA, 1990

[Covington, 1989] Michael A. Covington. *Efficient Prolog: A Practical Guide.* Research Report AI-1989-08, Artificial Intelligence Programs, The University of Georgia, August 1989

[ISO/IEC 1993] *Prolog — Part 1, General Core, Committee Draft* ISO/IEC JTC1 SC22 WG17 N110, International Organisation for Standardization, National Physical Laboratory, Teddington Middlesex, England, March 1993

[Maas, 1993] Christoph Maas. *Graphentheorie und Operations Research für Studierende der Informatik.* Wißner, 1.Auflage, Augsburg, 1993

[Shachnai & Yu, 1995] Hadas Shachnai, Philip S. Yu. *The Role of Wait Tolerance in Effective Batching: A Paradigm for Multimedia Scheduling Schemes.* IBM Research Report, RC 20038 (88607), NY, 1995

[Siegert, 1991] Hans-Jürgen Siegert. *Simulation zeitdiskreter Systeme.* Oldenbourg Verlag, München, 1991

[Storm, 1988] Regina Storm. *Wahrscheinlichkeitsrechnung, mathematische Statistik und statistische Qualitätskontrolle.* VEB Fachbuchverlag, 9.Auflage, Leipzig, 1988

[Wielemaker, 1996] Jan Wielemaker. *SWI-Prolog 2.7 Reference Manual.* University of Amsterdam, 1996