University of Karlsruhe (TH)
Faculty of Computer Science
Industrial Applications of Informatics and Microsystems (IAIM)
Prof. Dr.-Ing. R. Dillmann

Australian National University
Research School for Information Sciences and Engineering
Department of Systems Engineering

"Distributed temporal event mapping and fusion"

Diploma thesis

Felix Schill

01. April 2002 – 30. September 2002

Referees:	Prof. Dr.-Ing. R. Dillmann (IAIM)
	Prof. Alex Zelinsky (RSISE)
Supervisors:	Dr. rer. nat. Uwe R. Zimmer (RSISE)
	Dipl.-Ing. T. Asfour (IAIM)

Abstract

Localisation and mapping relies on the representation and recognition of features or patterns detected in sensor data. An important aspect is the temporal relation between observations in multiple sensor data streams. This thesis proposes a new approach for simultaneous localisation and mapping of temporal relations between observations.

Sensor data is interpreted as a sequence of events, where an event is the occurence of a distinguishable feature in the sensor data stream. The goal of this work is to develop a method for finding spatio-temporal correlations between those events from different sensor modalities, which occur simultaneously or which are reproducibly in causal order. Those correlations should be identified in an unsupervised learning process, represented in a suitable manner, and recognised efficiently. It also should be investigated, how temporal information can be used for localisation and mapping in a robotic context.

A dynamical system is proposed to acquire correlations between simultaneous and sequential events from different sources, to map causal sequences considering time spans, and to recognise previously observed patterns (localisation). The proposed method combines sensor modalities with different characteristics and timing behaviours, and is suitable for distributed computing. All sensor data streams are assigned to symmetric processes. For each sensor, mapping takes place locally in the assigned process. To achieve sensor fusion, all local maps are dynamically linked in realtime using direct inter-process communication. Mapping and localisation take place simultaneously in an infinite unsupervised distributed online learning process.

The dynamical system has been implemented as a distributed realtime system with symmetric processes. All processes communicate over ethernet. The final implementation uses the UDP protocol. The processes can be located on any computer connected to the robot's network. A realtime clustering network reduces the dimension of raw sensor data; cluster transitions are used as input for the dynamical mapping system. The functionality of the method has been evaluated in experiments with artificial and real sensor input data. Results from several physical experiments with different sensor configurations are presented.

Acknowlegdements

This thesis is the result of my visit at the Department of Systems Engineering at the Australian National University in Canberra. I would like to thank my supervisor Uwe Zimmer for the excellent support. I would also like to thank Alex Zelinsky, John Moore, Jochen Heinzmann, the staff and the members of the Department of Systems Engineering for inviting me, helping me organising my visit, and supplying support, a working environment, and all the ressources I required.

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbstständig und nur mit zulässigen Hilfsmitteln angefertigt habe. Alle benutzten Hilfsmittel sind im Literaturverzeichnis vollständig angegeben.

Karlsruhe, den 23. 9. 2002 Felix Schill

Zusammenfassung

Bei der Kartographierung und Positionsbestimmung kommt es darauf an, Merkmale und Muster in den Eingabedaten zu finden, geeignet abzulegen und sie wiederzuerkennen. Ein wichtiger Aspekt ist das zeitliche Verhältnis zwischen bestimmten Beobachtungen in mehreren Eingabekanälen. In dieser Diplomarbeit wird eine neue Methode zur simultanen Kartographierung und Positionsbestimmmung vorgestellt. Das Verfahren berücksichtigt und verwendet dabei zeitliche Verhältnisse in der Abfolge von charakteristischen Ereignissen in den Sensorkanälen.

Es handelt sich dabei um topologische Kartographierung im Raum der uninterpretierten Sensordaten. Die Sensordaten werden als Sequenz von Ereignissen interpretiert. Ein Ereignis ist das Auftreten eines unterscheidbaren Merkmals im Datenstrom. In dieser Arbeit geht es darum, eine Methode zu entwickeln, mit der es möglich ist, zeitliche Korrelationen zwischen jenen Ereignissen zu finden, die reproduzierbar gleichzeitig auftreten, oder in kausalem Zusammenhang stehen. Diese Korrelationen sollen unüberwacht identifiziert, geeignet abgelegt und effizient wiedererkannt werden. Es soll ausserdem untersucht werden, inwiefern zeitliche Information zur Positionsbestimmung genutzt werden kann.

Zu diesem Zweck wurde ein dynamisches System entwickelt. Dieses dynamische System ist in der Lage, in mehreren Eingangskanälen kausale Zusammenhänge unter Berücksichtigung der zeitlichen Abfolge festzuhalten, und bereits vorher beoachtete Muster wiederzuerkennen. Es kann auf Sensormodalitäten mit unterschiedlichem zeitlichen Verhalten und Eigenschaften angewandt werden, und ist besonders geeignet, als verteiltes System implementiert zu werden. Hierzu werden die Sensor-Datenströme symmetrischen Prozessen zugeordnet. Für jeden Sensor wird von dem entsprechenden Prozess eine lokale Karte erstellt. Alle lokalen Karten werden dynamisch in Echtzeit verknüpft. Kartographierung und Positionsbestimmung finden gleichzeitig und in Echtzeit statt. Es handelt sich um einen ununterbrochenen, unüberwachten Lernprozess.

Das dynamische System wurde als verteiltes Echtzeitsystem mit symmetrischen Prozessen implementiert. Zur direkten Prozesskommunikation wird UDP eingesetzt. Die Prozesse können auf beliebigen Computern laufen, die über ein Netzwerk verbunden sind. Zur Dimensionsreduktion der Eingabedaten wird ein dynamisches Echtzeit-Clustering-System *eingesetzt. Übergänge zwischen Clustern werden als Ereignisse betrachtet, und dienen als Eingabedaten für das dynamische System. Die Funktion des Systems wurde experimentell untersucht. Mehrere Experimente mit simulierten und echten Sensordaten wurden durchgeführt. Die Ergebnisse zahlreicher physikalischer Experimente mit verschiedenen Sensorkonfigurationen werden am Ende dieser Arbeit präsentiert.*

Contents

Chapter 1

Introduction

When regarding the localisation and mapping problem in mobile robotics, one approach is to geometrically interprete sensor data, and to create a kind of cartesian map ([5] and [6]). In some cases, though, a geometrical or any other interpretation of the sensor data is not clear, or not possible at all. Here mapping techniques working in sensor space have a big advantage, since they are still able to use the provided information. An example is an autonomous submarine robot: It is possible to use laser and sonar distance sensors and computer vision for localisation, but these sensors are very dependent on water conditions such as visibility, the amount of solved salt and suspended particles, water pressure, temperature and other parameters. While it is very difficult to consider all this information in a geometrical model, it is not necessary to consider it in a topological sensor space model of the environment, as long as local conditions are reproducible.

When working in sensor space, a great number of sensor measurements can be used, which would be useless otherwise. Water temperature, electric conductivity or the received light spectrum are just some examples. Of course, there are some drawbacks when mapping in sensor space. It is not easily possible to extrapolate the map for spots which have not been visited yet, but have been in sensor range. Here, the geometrical approach has a clear advantage. But there are many cases where geometrical information is not available, and localisation can only be done in sensor space. In a practical system, a combination of both approaches would be appropriate. Topological mapping usually has a lower resolution in space than geometrical approaches, but has less problems when dealing with large scale maps. Where precision is needed, a local geometrical map can be used and integrated into the topological world model.

The topic of this work is to develop a method for topological simultaneous localisation and mapping working in sensor space.

1.1 The role of temporal information

The context of a location plays an important role in the localisation and mapping problem. There are cases where no fixed reference frame is available, and movements in space cannot be detected reliably. This is usually not a problem while moving around on solid ground, but becomes obvious under water, or in the air. Good examples are submarines, hovercrafts, or flying objects such as planes, gliders, or balloons.

In those cases, time is the only reference frame for the context of a location which is left. The basic assumption behind this is a relation between the location and the behaviour of the vessel. In most robotic applications requiring localisation and mapping, this is given, at least in small scales. Usually this is sufficient to provide a stable temporal context.

The problem is how to represent time in a map. In the localisation and mapping context, time is not just the fourth dimension added to a three-dimensional map. The model of a four-dimensional space time is often useful in physics, but doesn't make sense for localisation. The big difference is, that a robot usually can move around freely (more or less) in three-dimensional space. But the movement in the time domain is predefined, unidirectional and cannot be influenced. The consequences are clear: If a location is defined in four-dimensional space, a robot could never return to the same location. Since localisation usually boils down to recognising a previously visited location in the map, this would make a localisation and mapping system obsolete. Mapping would be reduced to recording the input. Since no location could be visited twice, there is no possibility to generalise, or to close loops. Obviously, time can only be considered relatively, as time spans, or durations.

In this work, a location is defined as the position in three dimensional space, the orientation, the direction and current speed of movement. Since a robot is always moving through space on continuously differentiable trajectories, a location is correlated with a temporal context of locations along the current direction of movement. Assuming a sensor delivers a data stream which is correlated in some way with the actual location of the robot, it is possible to estimate the location from the lapse of sensor measurements over time.

A sensor stream can be abstracted to a sequence of events, where an event is the occurence of a transition to a previously observed similar sample of data. This transformation to an event sequence is a sampling process with an adaptive sampling frequency. The time span between samples is controlled by the amount of change in the input data. The maximal difference to the original data can be easily controlled by the distance measure used.

The task is now to identify sequences in the input stream, and to find out if they are reproducibly correlated to the location, and to their context. While it is easily

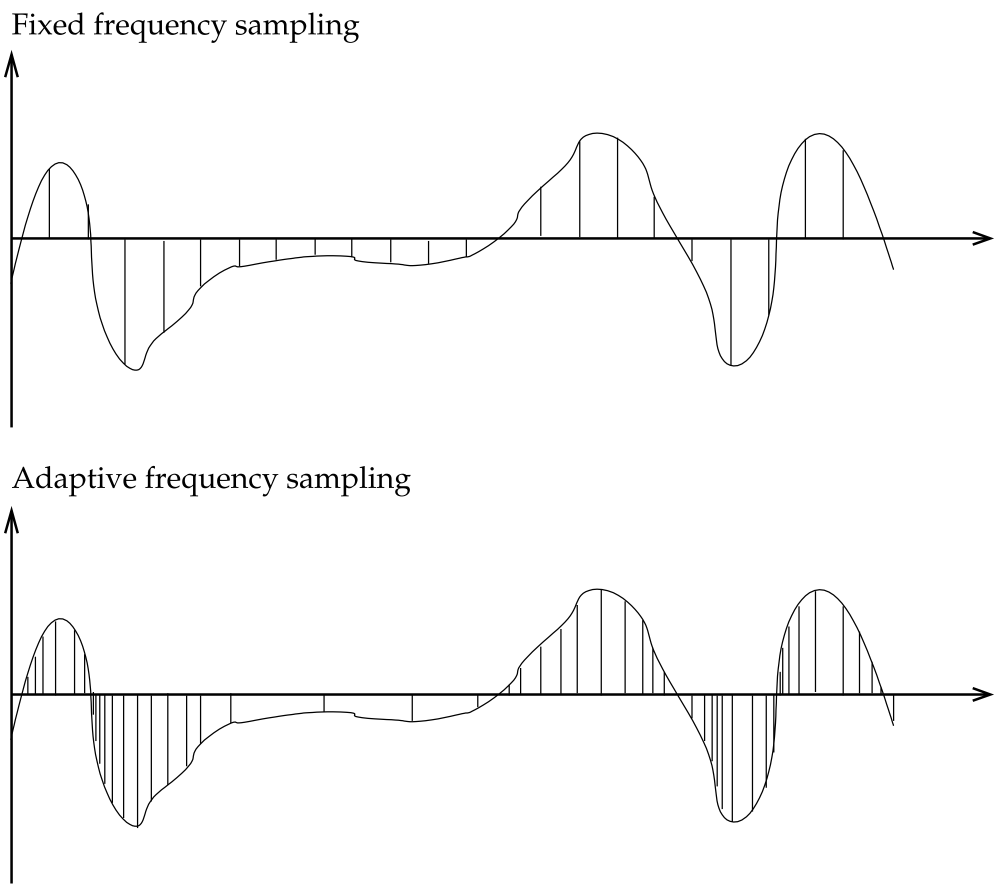

Figure 1.1: Illustration of data sampling with fixed frequency (above) and adaptive sampling.

possible to analyse the correct order of events (an automaton can do this), it is not so obvious how to recognise temporal patterns or rhythms. This is the main topic of this work.

1.2 Time sequence analysis for localisation and topological mapping

It is assumed that temporal information of input streams provides a lot of information about the current location, especially in environments with few features. The goal is to develop a mapping system, which identifies and uses spatio-temporal information in raw sensor data streams, in an unsupervised life-long learning process. It should be suitable for online operation in realtime environments. The emphasis is on considering temporal relations in sensor data streams, and finding correlations in data streams originating from multiple sensors with different characteristics. To remain as flexible as possible, as few assumptions as possible about the sensor characteristics, and no specific interpretation of sensor data should be used in the mapping system.

A single measurement at a discrete point in time only gives a faint idea about the location, since the dimensionality of reality is always reduced by the sensor. This creates ambiguities. Since a location is bound to its local context, time sequence analysis of the measurements can improve the estimation of the current location in space. Obviously it is impossible to be absolutely sure about the own location. Considering the recent context, and using all available sensors, it is possible to increase certainty to a sufficient level.

1.3 Overview: Existing works

There are several approaches to consider temporal information in the given input. An overview can be found in [2].

In [3], Barreto et al. present a neural network design which is able to learn and distinguish spatiotemporal sequences. The artificial neural network is used to track robot trajectories, and to learn and reproduce motion patterns. Their approach requires a dedicated learning phase, and is not able to automatically learn new inputs online. Also the amount of required neurons must be known in advance. It is not easily possible to let the network grow dynamically, to adapt to growing complexity of the input.

Euliano and Principe [4] extended the Self-Organizing Map and the Neural Gas with a Temporal Activity Diffusion mechanism (SOMTAD and GASTAD). As most

Self-Organizing Map models, the size of the map must be known in advance, and cannot be easily changed lateron.

Most methods were not designed for online and life-long realtime learning. The number of features in the input or the number or required neurons must be defined in advance. A common feature of those approaches is the required rigid timing of input streams, even if they originate from different sensors. That means that all input streams have to be resampled to a global sampling rate. This drastically reduces the possible temporal resolution in faster sensors, and often creates jitter effects and temporal artefacts. The other problem is, that a fast, high-bandwidth sensor observing a static environment, doesn't provide much new information in successive measurements. It would be appropriate to adapt the sampling rate to the complexity and dynamics of the measured data.

A biological method to deal with spatio-temporal patterns can be found in the cerebellum. In [1], it is assumed that the recognition of temporal patterns is based on 'tidal waves' of activity travelling along parallel delay lines. The input releases pulses on parallel axons. If (and only if) those pulses on neighboured axons arrive at a detector cell (called 'Purkinje cell') in a synchronized manner, they can stimulate this Purkinje cell. Temporal relations in the input are mapped to length differences on the parallel delay lines.

The basic idea of represent temporal patterns as lengths of delay lines and recognising patterns by detecting 'tidal waves' of activity was an inspiration for the model presented in this work. The proposed model is not similar to a biological cerebellum, and doesn't claim a biological background. The 'activity wave' idea and the concept of adaptable delay lines has been developed and adapted to meet the requirements of the localisation and mapping problem in robotics, and the possibilities of digital computers. The following chapters describe the design and functionality of this new approach.

Chapter 2

Multiple Hypotheses Temporal Mapping

This chapter proposes a method for topological mapping and pattern recognition in the time domain. At first a theoretical continous asynchronous analog model is presented, to explain the basic features and properties of the proposed system. The end of this chapter describes a similar model suitable for execution on digital computers.

2.1 Continuous analog model

The following description of the analog model should just simplify the explanation of the basic structure and functionality of the system, since it is more intuitive to describe. A precise definition and analysis of the analog system would go beyond the frame of this thesis. New analytic methods to analyse the system behaviour would have to be developed. In addition, it is impossible to implement the analog system with currently known technology. Thus all experiments, the analysis and the evaluation were done using the discrete model, described in 2.5.

The proposed model is a continuous asynchronous analog dynamical system. The input can be any n-dimensional continous analog sensor data stream. The system is able to map and to recognise reproducible temporal patterns in the input stream. The basic elements of the system are simple analog computation units, referred to as receptors, and delay edges connecting those receptors.

A receptor has an input for the sensor signal, and is permanently connected to the system's sensor input. For ease of description it is assumed that every receptor has immediate access to the current sensor data. Apart from the sensor data input channel, a receptor has one-dimensional input channels I_k, and one output channel O, which is also one-dimensional. The output is referred to as the activity level α of the receptor.

Receptors are connected by delay lines (or delay edges) $E = (\delta_E, \sigma_E, \omega_E, \lambda_E)$, which transmit the activity level of a receptor's output to an input of another receptor.

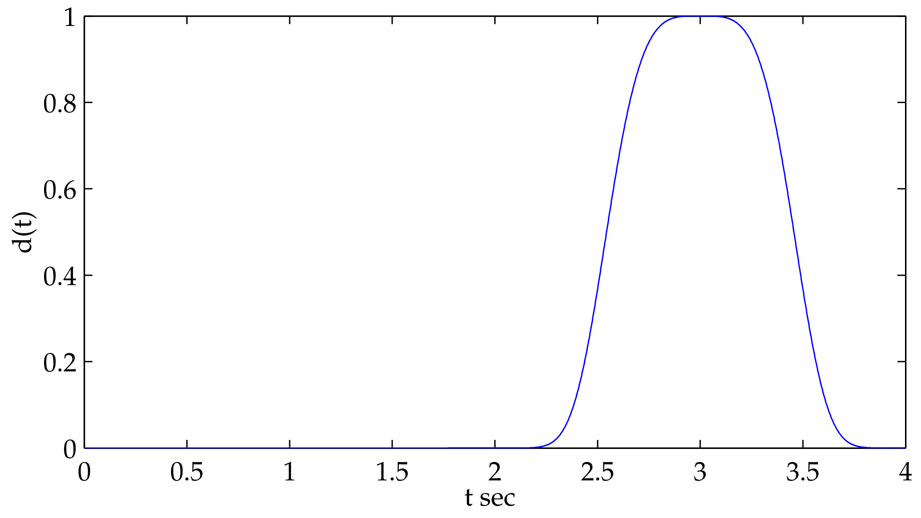

Figure 2.1: Plot of temporal transfer function $d_E(t)$ for $\delta_E = 3$, $\sigma_E = 0.5$ and $\omega_E = 1$

The transmission of the signal takes a defined, but adjustable time span δ. The signal is also damped during transmission; this is described by the adjustable parameter ω, called "edge weight". While travelling along the edge, the signal is widened to some extent, which is described by the parameter σ. The transfer function of a delay edge is as follows:

Definition 2.1.1 *Temporal function $d_E(t)$*

$$
\begin{aligned}
d_E : \mathfrak{R}^+ &\rightarrow [0,1] \\
d_E(t) &\mapsto \omega_E \cdot e^{\left(\frac{-(t-\delta_E)}{\sigma_E}\right)^4}
\end{aligned}
\tag{2.1}
$$

This function is symmetrically centered around δ_E; σ_E describes the tolerance or the width of the function. These two parameters are set and modified by the learning algorithm, as described later in 2.3.

A delay edge also has a "learning rate" factor λ, which describes the ability of the edge to adapt its parameters to other values.

A receptor can have an arbitrary number of incoming and outgoing delay edges. All outgoing edges are fed with the same activity level, while each incoming delay edge has an own input channel. Receptors and connecting edges are created online while learning temporal patterns, as described later in 2.3.

A receptor R holds a reference sample of sensor data, which it compares to the actual sensor data. This results in a similarity measure $s_R(t) \in [0,1]$. The basic functionality of a receptor is to select the input edge E_i with the maximum activity α_{E_i}, to amplify this activity level, and to send the amplified signal to the output.

The amplification function $amp(s_R(t)) \in [1/c_{damp}, c_a]$ is controlled by the similarity measure $s_R(t)$. The constant $c_{damp} > 0$ describes the maximal damping, $c_a > 0$ the maximal amplification of the function. The function should have steep flanks to narrow the shape of an activity wave going through a receptor.

The amplification function used in the later implementation has an impulse response of c_a, when the similarity measure enters an environment centered around 1, and is $1/c_{damp}$ otherwise. The size of this environment is controlled by a precision parameter p.

Definition 2.1.2 *Amplification function $amp(s(t))$*

$$amp(s(t)) \mapsto \begin{cases} c_a & \text{if } s_R(t) = p \wedge s'_R(t) > 0 \\ 1/c_{damp} & \text{otherwise} \end{cases} \tag{2.2}$$

To simplify later descriptions, the moment when the amplification function of a receptor R reaches the maximum of c_a shall be called “firing” of receptor R. “Firing” is a point in time and has no duration.

The output activity level of a receptor is limited by a saturating transfer function of the amplifier; the maximum output can be assumed to be 1. It is also assumed that each receptor has a minimum activity level $\min(\alpha_R(t)) > 0$, which is necessary to activate the system. This is controlled by the global parameter ε.

Definition 2.1.3 *Receptor activity: Let E be the set of all input edges of a receptor R. The receptor activity $\alpha_R(t)$ is calculated according to*

$$\alpha_R(t) = \phi\left(amp(s(t)) \cdot f_R(t)\left(\varepsilon + \max_{E_i \in E}\{\alpha_{E_i}(t)\}\right)\right) \tag{2.3}$$

ϕ is a saturating function, i.e. $\phi(x) = 1 - e^{-x}$

The function $f_R(t)$ describes the available firing energy of the receptor. It is zero after a receptor fired, and exponentially reloads to 1 afterwards (t_R is the last time receptor R fired, $E(t)$ is the number of events at time t since startup time t_0):

$$f_R(t) = 1 - e^{c_R(E(t)-E(t_R))} \tag{2.4}$$

2.2 The recognition process

The recognition of temporal patterns in the input stream is based on activity waves flowing through the network of receptors and delay edges. A pattern is represented by receptors holding a sample of the input at a certain time, and delay lines connecting those receptors. The delay lines hold the information about the temporal relation between the samples. Every activity wave represents a hypothesis about the current input. The location of a wave in the network describes the current input.

Assume there is already a network of receptors, representing a pattern in the input stream. Also assume the activity level of all receptors is minimal, and now the represented pattern occurs again in the input. This pattern will now shortly increase the amplification factor in the corresponding receptors, in a certain rhythm. The first stimulated receptor R_1 will amplify the low activity level on its inputs (caused by the self-activation term ε), resulting in a slightly higher activity on its outputs. This higher activity signal is transmitted along the outgoing delay edges to other receptors. One of those is stimulated by the sensor input approximately when this (widened) signal arrives. The signal is amplified again, transmitted to other receptors, and so on, until it saturates at the maximum activity level 1. A more accurate timing will result in a faster increase of activity and a higher over all activity level than inaccurate timing of events.

If the input differs from all represented patterns, the amplification factor of the receptors is low, when the activity wave arrives at their input. The activity wave is damped and disappears after a while.

It is obvious that several waves can appear and run simultaneously in the system, if parts of input patterns are similar. Those waves represent different hypotheses about the input. Nevertheless, the wave with the best matching receptors and delays will produce the highest activity, while other waves will decrease in activity or disappear completely.

2.3 Learning

Learning in this case has two aspects:

1. Learning new sequences by creating new receptors and edges (acquisition)
2. Modifying existing edges while recognising known sequences (adaptation)

The adaptation process runs continuously, the acquisition process is only activated if the current input stream is not recognized. There is no dedicated "learning phase". Acquisition and adaptation are a continuous life-long process, controlled by the input.

2.3.1 Aquisition

The acquisition process should obviously start when an input sequence is not recognised. Successful recognition of a sequence results in a high activity level

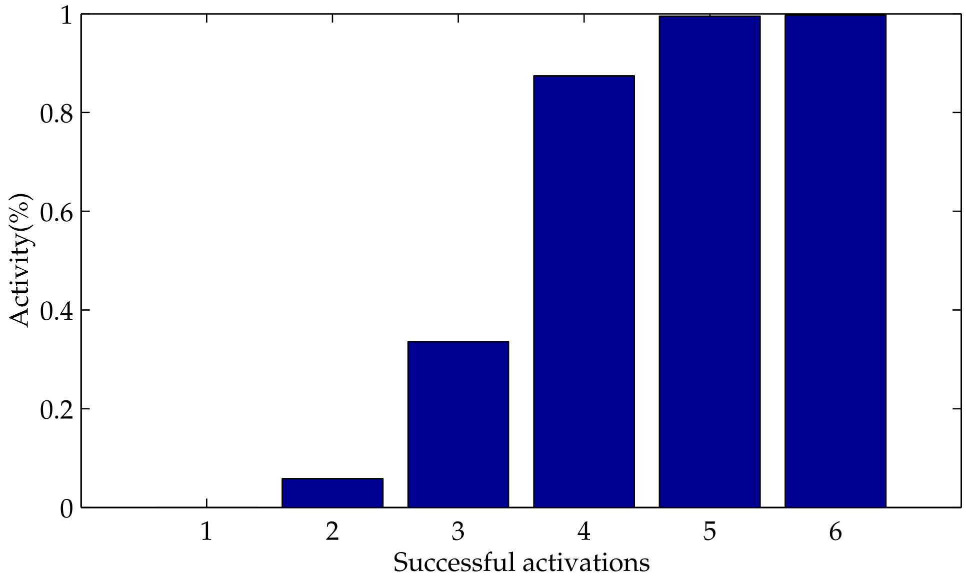

Figure 2.2: Boost of an activity wave ($c_a = 5, \varepsilon = 0.01$).

of the corresponding receptors. To distinguish between recognition and non-recognition, a threshold α_{high} shall be introduced. Additionally it is assumed that the most active activity wave in the system is known at all times.

If the similarity measure of the last most active receptor drops below the precision threshold p, and in the same moment, all other receptors have an activity level below α_{high}, the current sensor input is regarded as not recognised. In this case, the acquisition process starts.

A new receptor sensible for the current input is created. This is simply achieved by setting the internal reference sample of the new receptor to the current sensor input. The new receptor is connected to that receptor of the last most active activity wave, which most recently had an activity level higher than α_{high}. [1]

The delay δ_E of the connecting edge is set according to the observed time span between the last firing of the source receptor, and the creation time of the new receptor. It can be imagined that each receptor has an outgoing delay edge of infinite length, and the new receptor is connected to it where the maximum of the output pulse currently travels. The temporal tolerance σ_E is initialised with a fraction of δ_E, i.e. $\sigma_E := \delta_E/10$. The edge weight ω_E is initialised with a value lower than 1. This avoids that newly created edges dominate older and already stabilised edges. The initial value for the edge weight must be chosen high enough that an activation of the next receptor is still possible, though; that means it must

[1]This is described more precisely for the discrete model. Since the analog model cannot be implemented, it is difficult to describe this process.

be greater than $1/c_a$. In the experiments ω_E is initialised with 0.7. The learning rate λ is set to a plausible start value, i.e. $\frac{1}{2}$.

While the acquisition process continues, new receptors are created accordingly. Those are connected to the last newly created receptor, to represent the input signal over time. During this process, other hypotheses may emerge in the network. When an activity wave reaches the activity level α_{high}, the acquisition process stops, and the last newly created receptor is connected to the first receptor reaching an activity level greater than α_{high}.

Another aquisition mechanism fuses parallel hypotheses. Whenever two succesive most active receptors are not connected by a delay edge, the missing edge is inserted. If the edge already exists, it is adapted according to the adaptation rules described in the next section. This reduces the number of multiple similar hypotheses by fusing them. The adaption process will strengthen the most adequate path through the interconnected hypotheses, creating a suitable representation for all corresponding input patterns.

2.3.2 Adaptation

To consider variations of patterns in the input stream, adaptation of the internal parameters is necessary. Most information is stored in the delay edges. The sample of input data contained in receptors can also be adapted. This is not considered here, but dealt with in 2.5.2.

Whenever a receptor R_j fires and reaches a sufficient activity level ($\alpha_{R_j}(t) > \alpha_{high}$), the input edge $E_{R_i,R_j} =: E_{ij}$ with the highest edge activity level is selected. The selected edge is adapted using the following rules ($\Delta t = t - t_{R_i}$, t_{R_i} is the time when R_i most recently fired):

$$\delta'_E = \delta_E + \lambda_E \left(\tau_E \Delta t - \delta_E\right) \tag{2.5}$$

$$\sigma'_E = \sigma_E + \lambda_E \left(|\tau_E \Delta t - \delta_E| + t_{pmax} - \sigma_E\right) \tag{2.6}$$

$$\omega'_E = \omega_E + \lambda_E \left(1 - \omega_E\right) \tag{2.7}$$

$$\lambda'_E = c_\lambda \lambda_E \tag{2.8}$$

On good matches, the temporal tolerance σ_E is reduced, and the edge weight ω_E increases. The constant t_{pmax} expresses the maximal temporal precision, that means σ_E will not drop below this value. The duration parameter δ_E is adapted to get closer to the current observation. The reduction of the learning rate λ_E stabilises the process. The weights of the output edges of R_j are decayed by the constant c_d:

$$\omega'_E = \omega_E - \lambda_E c_d \omega_E \tag{2.9}$$

This effect is inversed in the next step for the edge which successfully activates a receptor. The weights of all other edges remain slightly lowered. Frequently occuring and successfully recognized sequences strengthen corresponding edge weights, while unused or inaccurate edges decay. This results in a probality representation coded into the edge weights, which enables the system to consider statistical information.

2.4 Model characteristics

The presented model is an asynchronous analog parallel system, which samples the input with a variable, sensor data driven time basis, and connects these samples to a graph representing causal and temporal correlations. Recognition of known input patterns is based on resonance of an input with sequences of receptors in the network, connected with delay edges. This resonance behaviour is realised with activity waves travelling through the network, being amplified or damped depending on the input.

The resonance behaviour is (apart from the sensor input) mainly controlled by the amplification function $amp(s(t))$ (the parameters c_a and c_{damp}) and the self activation parameter ε. A high amplification term leads to steep growth of activity waves. The parameter ε controls the startup time of a wave.

Since there is no rigid time basis, the system is able to cope with any speed of change in the input, as long as not limited by hardware constraints. The sampling precision in the acquisition process is controlled by the similarity measure $s(t)$ and the threshold p.

The immediate acquisition of input data enables the system to learn new input patterns extremely fast. The adaption process improves the recognition by considering the temporal variance and probability distribution of similar patterns.

2.5 Discrete model for digital computers

An analog implementation of the mapping system would have great advantages. Unfortunately it is impossible to implement the system with currently available analog technology. No technique is known to implement fully adjustable analog delay edges, which are able to produce delays ranging from microseconds up to several seconds or even longer. It is also extremely difficult to implement a dynamical creation of receptors and edges in a hardware system. To solve these problems, a similar model suitable for execution on digital computers is presented in this section.

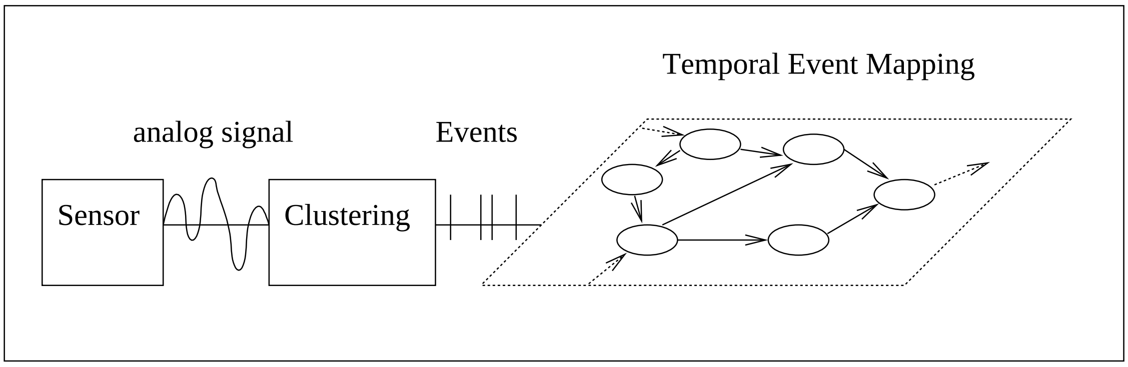

Figure 2.3: The basic layout of the discrete mapping system

2.5.1 General problems

Obviously it is impossible to achieve exactly the same behaviour as in the analog counterpart. But since most sensors used in robotics provide discrete samples with a fixed frequency, a discrete processing of this data is adequate and legitimate.

One problem is the execution of the massively parallel analog system on a single digital processor. Another problem is the simulation of a continous process on a sequential machine. To solve these problems, the input sensor data is discretised by a clustering technique described in the next section, and discrete events in time are introduced. This allows the calculation of receptor and edge activities at a definite time.

2.5.2 Clustering / data preprocessing

To reduce the dimension of the input data, a dynamical online clustering technique from [7] is applied. The clustering system is a modified neural gas algorithm, but without any static restrictions. A network

$$\begin{aligned} Net &= (C, E, R, N, \alpha) = \\ & (c_i, \\ & (c_j, c_k) \in [0, 1], \\ & R(c_i) \in \Re^n, \\ & \alpha(c_i) \in [0, 1]), i \in 1..n \end{aligned} \tag{2.10}$$

is created online, representing the clusters identified in the input data. It consists of n cells C, edges E, representatives R (the 'centers' of the data space covered by one cell), and an adaptation parameter α.

Starting with an empty network, new cells are created whenever an input sample S_t doesn't match the representative $R(c_i)$ of any cell in the network. If a match was found, the cell parameters and edges are adapted. For comparing input samples with representatives, a metrics $\| \bullet, \bullet \|$ is defined; the resulting distance is compared to a precision parameter p.

To meet realtime constraints, the network is only searched and adapted locally, until a match was found. The typical execution time after stabilisation is low and more or less constant. Only unknown input or rare transitions cause a slightly longer execution time for a complete search and reorganisation of the network. Experiments show that the clustering system works stable and can handle input streams from commonly used sensors in realtime. For the localisation and mapping approach in this work, the ability of the network to grow dynamically, was considered to be more important than hard realtime constraints. Graceful degradation is acceptable for this application.

Using the described clustering network, a multi-dimensional sensor data sample is reduced to a one-dimensional cluster identifier.

2.5.3 Discrete events

Instead of feeding a continous sensor data stream into all receptors, it simplifies calculation a lot to define points in time when certain receptors should be evaluated since they are likely to fire. In the analog model a receptor decides itsself if to become active, depending on the current sensor input. In combination with the clustering technique, this can be separated completely from the receptors themselves. A transition between clusters in the clustering system is a significant occurence, which shall be called event.

An event happens at a discrete point in time, and is described by the identifier of the cluster which became active with the occurence of the event. The average event frequency will usually be significantly lower than the sensor frequency (which is the upper bound for the event frequency). The time span between two events can be any multiple of the time span between two sensor data samples, and depends on the variations in the sensor data stream.

Instead of receptors with individual samples of data, there are now sets of receptors sensible to a certain event class (cluster identity). Now only when an event occurs, all receptors of its class have to be evaluated. This slightly varies from the firing behaviour of a receptor in the analog system, but this doesn't affect the functionality of the system. The difference is that now several receptors share the same sample of input data, instead of carrying an own sample each. The recognition process is not affected by this, since it could happen in the analog model as well, that several receptors carry identical sensor data samples. The resulting graph structure is slightly different, but is still a valid representation of the corresponding input data. Solely the sample points in time vary negligibly.

2.5.4 Calculation of receptor activity

On an occurence of an event, all receptors sensible to its event class are evaluated. Since the comparison of the current input data with a sample has already been done in the clustering network, the similarity measure is assumed to be 1.

The edge activity of incoming delay edges has to be calculated using the temporal transfer function, and the time when the last pulse has been sent over that edge, that is the last activation time of the receptor at the origin of this edge.

Definition 2.5.1 *Edge Activity: Let E_{ij} be a delay edge leading from receptor R_i to R_j. The edge activity $\alpha_{E_{ij}}(t)$ is calculated according to*

$$\alpha_{E_{ij}}(t) = d_{E_{ij}}(t - t_{R_i})\,\alpha_{R_i}(t_{R_i}) \tag{2.11}$$

(t_{R_i} is the time of the last activation of R_i)

The receptor activity is calculated as defined in the analog model (definition 2.1.3). Since this calculation only takes place when a receptor fires (i.e. the sensor data enters the cluster around the data sample of that receptor), the amplification function $amp(s(t))$ can be replaced by the maximum c_a.

The activity of a receptor R_i is now coupled to the discrete point in time when the triggering event occured. This time is stored in the receptor as the time stamp t_{R_i} of the last activation.

2.5.5 Damping of activity waves

Since receptors not corresponding to incoming events are not evaluated, damping of activity waves is not done by now. Instead, waves are lost completely when an event does not occur as expected. To implement the damping feature of the analog model in the discrete model, a bit of extra effort has to be done. The following approach solves the problem.

Whenever a receptor is activated, all receptors connected to its output are added to a list, together with the times the output pulse will arrive at their inputs (these time spans are stored in the corresponding delay edges).

A separate task waits until the next pulse arrives at a receptor in this list (the time can be read from the list). If the event triggering this receptor has not yet occured, the output activity of this receptor is calculated, using the damping factor c_{damp}, instead of the amplification factor c_a. This is equivalent to a similarity measure $s(t) = 0$. Again, the receptors connected to the output are added to the list, provided the output activity level is not too low.

This process will continue all activity waves in the network, damping them, until their activity is below an "activity noise level" α_{low}. If an event should trigger a receptor after the damping process, the output activity is automatically overwritten with the new value. This leads to the desired behaviour, that a wave is amplified if the expected event occurs in time, and damped otherwise.

To ensure consistency of the graph data structure and the states of receptors, mutual exclusion of input data processing and damping of activity waves has to be guaranteed. Whenever new data arrives, the task processing the damping of ongoing activity waves is suspended, but only after it finished its current work. When calculations for the newly arrived data are finished, this task is notified and continues processing.

2.5.6 Learning

Learning works similar to learning in the analog model (section 2.3). The only difference is the start criterion for the acquisition process. This is easier to define in the discrete model, since activity is now partially synchronized to events. Whenever an event occurs, and, after evaluation of all corresponding receptors, none of them reaches an activity level above α_{high}, the input can be regarded as not recognised. This triggers the acquisition, which stops after an event occurs and one of the already existing receptors reaches a high activity level.

2.5.7 Time warping

The model can easily be extended to be able to recognise known sequences even if they occur faster or slower. This effect is called time-warping. Therefore a time-warping factor is introduced, controlling the speed a signal travels on the edge. The calculation of edge activity as defined in definition 2.5.1 changes as follows:

Definition 2.5.2 *Edge Activity with time-warping: (E_{ij} is a delay edge from receptor R_i to R_j)*

$$\alpha_{E_{ij}}(t) = \omega_{E_{ij}} \cdot d_{E_{ij}}\left(\tau_{E_{ij}}(t)(t - t_{R_i})\right)\alpha_{R_i}(t_{R_i}) \tag{2.12}$$

The time-warping factor $\tau_{E_{ij}}$ is dynamically adapted during the recognition process. On activation of receptor R_j, the best matching input edge (R_i, R_j) with maximum activity is selected, and τ' is adapted to maximize $d_{E_{ij}}(\tau' \cdot (t - t_{R_i}))$. The new time-warping factor $\tau_{E_{jk}} = \tau_{E_{ij}} + c_t(\tau' - \tau_{E_{ij}})$ is propagated to all output edges E_{jk} of R_j (c_t controls the speed of the time warp adaptation). If a receptor doesn't reach a sufficient activity level, the time warping factor is reset to 1.

2.5.8 Complexity and realtime constraints

The temporal mapping system dynamically grows to meet the spatio-temporal complexity of the input data. The typical calculation time per sample of the clustering system linearly depends mostly on the sensor data dimensionality n. The maximum complexity is $O(nk)$ for k clusters. The complexity of the temporal event mapping network depends on the number of receptors per cluster. The average complexity thus is $O(|R|/k)$. The problem is, that both the number of clusters k and the number of receptors $|R|$ grow dynamically, to meet the intrinsic spatiotemporal complexity of the input data.

In a limited environment, the whole system stabilises after a while, which means that the number of clusters and receptors remains constant. So if the available processing power is chosen according to the expected complexity of the target environment, realtime operation is possible. In the experiments described later (5.3), the system stabilised with ususally 50-70 clusters, and 400-500 receptors. Using an Intel Pentium III system with 633Mhz and a SICK laser sensor with 5 scans per second as input, this leads to typical average execution times per event of 180 ms for the clustering network, and 3-5 ms for the temporal mapping system. The sensor frequency of 5 Hz could be maintained most of the time. It should be mentioned that the occasional loss of few samples is not critical towards the functionality of the mapping system (graceful degradation).

Chapter 3

Sensor fusion and Distributed Temporal Event Mapping

The model described in the previous chapter is able to map temporal correlations in one input stream into a topological map. This chapter extends the single sensor model to cope with multiple sensor modalities. The goal is to achieve more stability by using and correlating input data from all available sensors, where this is possible. This is realised by mapping temporal and causal correlations between different sensors in a similar way as it is done for each single sensor.

3.1 Extension to multiple sensor modalities

Let S be a set of arbitrary sensors S_ψ, which produce n_ψ-dimensional data streams. The basic idea for the extension is to apply the single-sensor system to each available sensor S_ψ, and then cross-correlate causalities and temporal relations between the sensors. Again, this process is fully sensor-data driven.

In the first step, the dynamical systems for all sensors run independently, building up graphs representing the transitions of input data. For interconnection of different sensor modalities, the same basic mechanisms from the single sensor model can be used. To represent a temporal relation between patterns in different sensors, the corresponding receptors in both systems are connected with a delay edge similar to internal delay edges. The properties of these intermodal graph edges (referred to as “cross edges”) are identical to those of internal delay edges. Though, the creation and adaptation of cross edges follows different rules, and their influence on receptor activity has to be adapted.

3.1.1 Creation of cross edges

When a receptor R fires and reaches an activity which is higher than α_{high}, this means that some structure in the input stream has been recognised.

Whenever this happens, for each other sensor modality a limited set of the last most active receptors R_{lma} since creation of the last cross edge with an activity level greater than α_{high} is determined. The size limitation of the set R_{lma} is chosen depending on the average frequencies of involved sensors, and the expected influence of an event on following events in other sensors. The size of this set represents the "short term memory" of the system for external input, describing for how long an event is still considered to have influence on the current status. In the experiments, the set was limited to the 3 last most active receptors.

For each Receptor in R_{lma} a cross edge to R is created, using the difference of the maximum activation times as the delay. The other edge parameters except the edge weight are also initialised according to the rules for internal edges as described in section 2.3.1. The edge weight ω is initialised with a value close to zero, i.e. 0.1. This requires cross edges to establish before they can have influence on the system.

3.1.2 Adaptation of cross edges

The learning rules of cross edges are similar to normal edges (see 2.3.2). The only difference is, that not only the most active edge is adapted. Instead, all incoming cross edges are adapted, while the amount of adaptation is weighted by their output activity, not considering the current weight. This leads to the following slightly modified adaptation rules:

$$\begin{aligned}
\delta'_E &= \delta_E + \beta_E(t) \cdot \lambda_E \left(\tau_E \Delta t - \delta_E\right) && (3.1)\\
\sigma'_E &= \sigma_E + \beta_E(t) \cdot \lambda_E \left(|\tau_E \Delta t - \delta_E| + t_{pmax} - \sigma_E\right) && (3.2)\\
\omega'_E &= \omega_E + \beta_E(t) \cdot \lambda_E \left(1 - \omega_E\right) && (3.3)\\
\lambda'_E &= \beta_E(t) \cdot c_\lambda \lambda_E && (3.4)
\end{aligned}$$

with

$$\beta_E(t) = e^{\left(\frac{-\left(\left(t - t_{R_i}\right) - \delta_E\right)}{\sigma_E}\right)^4} \alpha_{R_i}(t_{R_i}) \qquad (3.5)$$

(see also definition 2.1.1 and section 2.3.2).

It is necessary to have a weighted adaptation for all edges in this case. For internal edges, the goal is to establish one path per hypothesis through the graph, to stabilise recognition and prediction ability. This is not the case for cross edges. Here it is important to find correlations, and to weight the influence depending on reproducability and accuracy. When using the "winner takes it all" strategy of internal edges for cross edge adaptation, it was observed that the input cross edge which could establish was not necessarily the most important one. Small advantages depending on the start conditions increased, so that it was sometimes rather random which edge was selected. Once ahead, winning was much easier for this edge, so it supressed all other edges. The weighted adaptation approach doesn't suffer these problems.

3.1.3 Modification of Receptor activity calculation

Since a receptor can now have inputs from different sensor modalities, the calculation of output activity has to be slightly modified. As before, the internal edge with the maximum edge activity is selected for activity calculation. The activity of all incoming cross edges is summed up and put through a saturating function, to limit cross edge influence. The result is added to the internal edge activity influence:

Definition 3.1.1 *Receptor activity for multiple sensors: Let* $E_{S_\psi} := \{E_{ij} = E_{(R_i,R_j)} : R_i \in R_{S_\psi}\}$ *be the set of all input cross edges of receptor* R_j *from sensor modality* $S_\psi \in S$*, and let* $S_I \in S$ *be the set of involved sensor modalities:*

$S_I := \left\{S_\psi \in S : E_{S_\psi}(R_i, R_j) \neq \varnothing\right\}.$

The receptor activity $\alpha_{R_j}(t)$ *is calculated according to*

$$\begin{aligned}\alpha_{R_j}(t) &= \phi\left(amp\,(s(t))\left(\varepsilon + \max_{E_i \in E}\{\alpha_{E_i}(t)\} + c_I \cdot \tanh\left(\frac{\gamma}{c_M}\right)\right)\right)\\ \gamma &= \sum_{S_\psi \in S_I}\sum_{E_{ij}\in E_{S_\psi}} \alpha_{E_{ij}}(t)\end{aligned} \tag{3.6}$$

(see also definition 2.1.3)

The constants c_I and c_M control the strength of influence from cross edges. The parameter c_I describes the upper limit of cross edge influence, and c_M describes the number of modalities with maximum input, for which the influence saturates.

3.2 Interaction between sensor modalities

Active receptors in different sensor processes can now influence each other. If events from different sensors occur in the same temporal order as they were previously observed, activity in the corresponding receptors is reinforced over the previously created cross edges. Hypotheses in each modality which consider external input are more likely to become most active. Uncertainties in one sensor modality can be eliminated or reduced now, using information from other modalities. The system is less sensible to noise, since the sensor modalities stabilise each other.

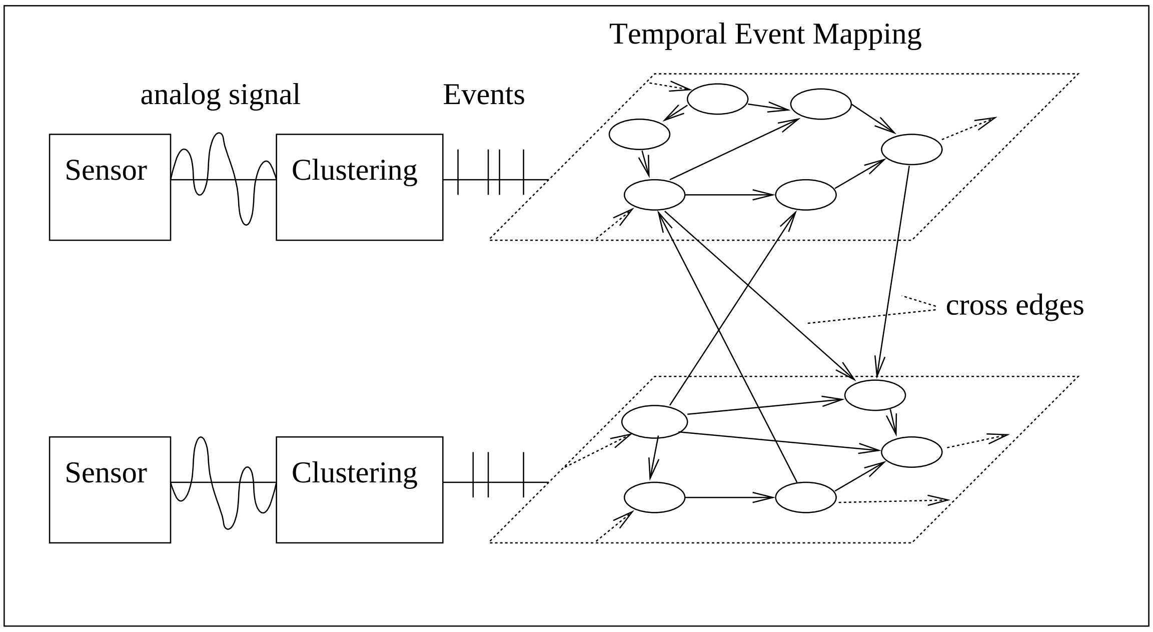

Figure 3.1: The distributed mapping system for multiple sensors.

Cross edges are initialised with a low edge weight. If a cross edge represents a real correlation, it will be strengthened by the adaptation process, since it successfully contributes to the activation of the target receptor. Otherwise, this cross edge decays. The system is able to detect and use temporal and causal correlations, which are reproducible, and ignores irreproducible noise.

The system behaviour of the multi-sensor model applied to only one sensor is identical to the behaviour of the single sensor model. Since cross edges are only created after a high activity level is reached, the startup behaviour of each single sensor in the multi-sensor system is also identical to the single-sensor case. After a stabilisation phase in each single sensor modality, the number of cross edges and the effects of sensor fusion increase. It is always possible to add another sensor modality to the system. It will integrate itself into the existing system after local stabilisation. A sensor can also be easily removed from the system by simply deleting all existing cross edges between this sensor modality and the rest of the system. It is even possible to temporarily deactivate a sensor by ignoring its cross edges during the receptor activity calculation. This flexibility is very useful for real systems, where sensors can fail, or are not necessarily available from the beginning.

3.3 Discrete model

The modifications of the analog model as described in the single sensor case (section 2.5) also apply to the multiple sensor analog model. Other modifications are not necessary.

The discrete multiple sensor system consists of symmetric partitions, one for each sensor. Splitting the whole application into symmetric partitions to be executed on different machines, holds many advantages. In real systems, the number of interfaces for sensors on one computer is limited, and so is the available processing power. A distributed mapping system makes less limitations to the number and geographical position of sensors. Every partition does the mapping for one sensor, which is locally accessible. The sensor fusion functionality, as described in this chapter, is achieved by communication over a network with the partitions for the other sensors, which are running on computers accessible over the network. This approach also enables the use of different platforms and architectures for each sensor in the same system. Depending on the sampling rate and data dimensionality of a sensor, a small microprocessor system might be appropriate for one sensor, whilst another sensor should be connected to a high performance computer. The only requirement is the availability of the desired network protocol and the compiler on each target platform.

This section discusses topics concerning the implementation of the discrete multi-sensor mapping system as a symmetrical distributed application.

3.3.1 Handling of intermodal graph edges (cross edges)

It is obvious that delay edges between sensor modalities running on physically separated machines have to be handled differently than internal graph edges. To reduce the amount of communication, and to keep the partitions as independent as possible, the following approach has been chosen.

Only active receptors have an influence on other sensor modalities. Thus it is sufficient to transmit information about active receptors to all other partitions. Due to real time constraints, it is necessary to limit the message size, so only the states of the n most active receptors are transmitted.

The activity of cross edges is needed for receptor activity calculation. It is very useful to keep all information about incoming cross edges on the receiver side. On sender side, no information about outgoing cross edges is necessary. When a message containing the state of the most active receptors of sensor modality S_ψ arrives, the receiver decides if to create cross edges. If a cross edge has to be created, a local copy of the corresponding source receptor from S_ψ is created. The cross edge can now be created and treated as an internal edge, since the source receptor is now in the same memory space. The state of the local copy of the remote receptor is regularly updated, whenever a message containing information about it arrives.

The activity calculation is timestamp oriented. The timestamp of the local copy has to be consistent to the local system time. One method to achieve this, is to synchronize the clocks of all participating machines, respective to calculate clock skews to be able to translate timestamps to the internal time. The clock synchronisation has to be repeated regularly, since clocks can drift during runtime.

Since messages about activation are always sent out immediately after an event occured, it is also possible to use the arrival time of a message as the timestamp. In this case, communication delays are mapped into the system together with the usual delays between events. This is not a problem, as long as they are more or less reproducible. Both methods are legitimate and yield similar results.

Chapter 4

Implementation and design issues

The distributed mapping system presented in the last chapter has high demands towards implementation. This chapter describes the design of the most important features, and the layout of the distributed system. A big issue is task synchronisation, and controlled access on shared dynamical data structures.

4.1 Programming language of choice

The described system requires support for realtime programming and multi-tasking. Especially task synchronisation mechanisms, mutual exclusion (monitors), controlled time-outs and guaranteed scheduling times are important requirements. It is clear that a temporal mapping system relies on quick response times, and a sophisticated handling of time in the programming language and the runtime environment. For development and debugging, features like exception handling and range checks at runtime are useful.

A language which meets most requirements is Ada95. This revision of Ada83 from 1995 extends the original Ada with object-orientation, and many loadable extensions (annexes) for realtime programming, distributed systems, communication and more. Ada95 has a built-in language support for multi-tasking, monitors, guarded and unguarded entries, task synchronisation, exception handling, flexible scheduling time handling (“delay until”-statement), and offers many possibilities for flexible type definitions. All types are checked during runtime. It is possible to define protected types or methods which can only be accessed by one task at a time (mutual exclusion). The realtime scheduler of the included runtime environment can be configured freely or even be replaced by an own implementation, if necessary. An open source compiler is available for most platforms.

Other common languages like C++ or Java are usually missing the flexible type definitions and the support for monitors, guarded task entries, and protected types. Since these are basic requirements for the implementation of the presented temporal mapping system, Ada95 is the language of choice for this project.

4.2 Tasks and monitors

Basically, there a two input sources. One is the event stream from the clustering system or the sensor. The other input is the communication system, receiving messages from other partitions. Both input interfaces should be designed as monitors, equipped with a non-blocking input entry, and a blocking data retrieval entry. Following processes (or tasks) are necessary in the mapping system itself.

The event input monitor. This monitor has a non-blocking "Notify" entry, to feed an event into the system. The event is buffered in the monitor, and retrieved by the main mapping task using the "GetEvent" entry. The "GetEvent" entry is a guarded entry, which blocks until a new Event has arrived, and the system is up and running. Another entry called "ShutDown" triggers the system shutdown of the mapping system. For the experiments with manually entered landmarks, another entry "SubmitLandmarks" receives externally entered landmarks and forwards them to the mapping system, where they are attached to the related receptors. With these entries, the event input monitor offers a complete input interface for the mapping system.

The main mapping task. This task blocks itself by calling the data retrieval entry of the event input monitor. Thus this task is fully event-triggered, and doesn't become active without incoming events. It is responsible for all calculations and procedures concerning the local mapping and recognition. Here, the activity levels of receptors are recalculated. Only this task can create edges and receptors, and it is responsible for edge adaptation.

The wave damping task. In the analog system, activity waves not supported by the current input do not disappear suddenly, but travel along through the graph for a while, being damped until they disappear. To achieve this behaviour in the discrete system, a separate task is necessary. This task is suspended while the main mapping task is active. After the main mapping task has finished its active phase, it notifies the wave damping task. This task calculates the time when an activity wave will reach the next receptor in the system, and waits until then. Now, the activity level of this receptor is recalculated using the activity level of the considered wave, and the damping factor. Again, the time is recalculated, when an activity wave reaches the next receptor.

This task has to store information about current activity waves in the system. To meet realtime constraints, waves are not continued anymore, after their activity drops below an insignificant level α_{low}. Since this task also accesses the main graph data structure, and reads and modifies receptor activity levels, mutual exclusion with the main mapping task has to be guaranteed. To achieve this, the main mapping task calls a blocking "suspend" entry of the damping task. The damping task completes its calculations, and is suspended. It only becomes active again, after the main mapping task has finished, and calls the "notify" entry of the damping task.

The communication task. This task obtains its input data from a blocking monitor entry of the communication subsystem. It becomes active, whenever a message arrives. To avoid data access conflicts or inconsistencies in the graph data structure, it only processes the incoming message and buffers all relevant data in a protected list. The main mapping task considers the information in this list, and creates or adapts cross edges if necessary.

This task also updates all local copies of remote receptors, for cross edge activity calculation. To guarantee data consistency, all local receptor copies are stored in protected list structures. This protected list provides atomic read and write access to list elements, and is used in several places of the implementation.

The communication input monitor. This monitor is responsible for the interaction of the communication subsystem, and the mapping task. It has been integrated into the communication subsystem. It offers a "Listen" entry for message retrieval. The "Listen" entry itself is non-blocking, but callers are requeued to a blocking waiting queue entry. There they are blocked until a message arrives. Like this, an arbitrary number of recipients can be accepted, which are all blocked until a message arrives. Then they all are released and obtain the current message. Messages are delivered to the monitor by the communication subsystem.

The main application task. This task is responsible for sensor data retrieval, sensor data preprocessing (clustering), and triggers the mapping system by delivering events to the event input monitor. It is not belonging to the mapping subsystem like the tasks and monitors described before, but runs externally. The mapping system itself is event-triggered, and does not run without event input. Usually, the main application task blocks itself at the sensor driver, waiting for new sensor input. In this case, it is the resposibility of the sensor driver to provide a continous input stream. If the sensor driver is passive, the main application task has to poll the sensor driver for new input.

All procedures accessing the graph data structure of the topological map are defined in a monitor. This guarantees atomic operations and data consistency.

4.3 Graph data structure of the topological map

The main data structure is the graph representing the topological map. The graph consists of receptors and edges, which are stored in separated data objects each. Both receptor and edge data objects have to be allocated dynamically. This enables the graph to grow dynamically during runtime.

4.3.1 Receptors

A receptor has to carry the following data:

ID $\in N$**:** A unique identifier. All identifiers in the implementation are numerical identifiers, stored in a long integer variable.

Triggering Event: The last event which activated this receptor most recently. An event is described by the time when it occured (timestamp), and the event ID. The event ID is usually derived from the corresponding cluster ID.

Activity: The output activity of the last activation. This information together with the event timestamp is necessary for the activity calculation of following receptors.

Event count $\in N$**:** This value is updated every time the receptor is activated. It is used to determine the number of events since a previous activation, to calculate the firing energy.

Input edges: A list of incoming internal edges. The list entries are only references to the edge objects, since edges are also accessed as output edges of the source receptor.

Output edges: A list of outgoing edges. Again, a list entry is only a reference.

Input crossedges: A list of modality descriptors with incoming crossedges. A list entry is a descriptor containing the modality identifier, and a list of references to crossedges from this modality.

Mirror flag $\in \{true, false\}$**:** This flag indicates if the receptor is a local receptor, or a local copy of a remote receptor in another modality.

4.3.2 Edges

An edge object contains the following data structures:

Edge parameters: The delay duration δ, temporal precision σ, learning rate λ and the edge weight ω ($\delta, \sigma, \lambda, \omega \in [0, 1]$). These parameters are described in chapter 2.

Time warping factor: The time warping factor $\tau \in \Re$ is forward propagated along the output edges during the recognition process.

Source: A reference to the source receptor of the edge.

Target: A reference to the target receptor of the edge.

Crossedge flag $\in \{true, false\}$**:** A flag indicating if the edge is a crossedge, or an internal edge.

4.4 Data layout of the system

Apart from the main graph data structure, some other constructs are required. These supplemental data structures are necessary for efficient access to the graph structure, and for the management of the distributed system.

4.4.1 Protected list

For concurrent access control on multiple data elements, a protected list structure is necessary. The list has been designed as a double-linked list. In this implementation, a list is described by a list descriptor. The list descriptor contains references to the head and the tail of the list (the first and the last element).

For all list operations, a list handle is required. A handle either specifies exactly one element in the list, or is outside the list and doesn't specify an element. A handle for a list can be created from a list descriptor with a protected procedure (CreateHandle).

The following operations on the list are required:

- Jump to head (moves the handle to the begin of the list)
- Jump to tail
- Step forward
- Step back
- Get current element
- Add (append to list)
- Insert element at current position
- Remove (removes the specified element from the list
- Overwrite (overwrites the specified element with a new value)

Additionally, for list management, the following operations are necessary:

- Create list (creates a list descriptor, defining a new list)
- Create list handle
- Duplicate list handle

- Release list handle

A status function IsValid checks if the list handle specifies an element, or points outside the list.

To guarantee data consistency with concurrent access on the list, a list element may only be accessed by one process at a time, and all operations have to be atomic. To achieve atomicity, all list access procedures have been defined in a protected monitor.

Additionally, a list element must not be removed if it is still referenced (there are list handles pointing to it). Every list element counts the number of existing references. The reference counters are updated by all procedures which are modifying list handles. The "remove" procedure marks the list element for deletion. Afterwards, it is not possible anymore to move a list handle to this element. List handles still pointing to an element marked for deletion, are still valid. As soon as the last list handle leaves this element, it is finally removed from the list and deleted.

It is important to release list handles after use. This means that they are moved outside the list. If this wouldn't be done, it could prevent list elements from being deleted.

To guarantee the correctness of the reference counter in list elements, list handles may only be created, modified and duplicated by special procedures, which are updating these counters. They should be declared as limited private types.

The list was implemented as a generic list. List elements are containers, which carry an object of the specified type. Deleted list element containers are collected in an internal recycling list, and reused for new elements. This avoids deallocation of memory, and thus avoids memory fragmentation.

4.4.2 Event association table

Incoming events have to be associated with their corresponding receptors. For this, a list is used, where the elements contain the event ID, and the list descriptor of the list with the corresponding receptors. When a new event arrives, the corresponding list element is identified, to get the list with corresponding receptors. All receptors in this list have to be evaluated.

If the incoming event cannot be found in the list, a new entry is created, and appended to the list. The event association table dynamically grows with the number of possible events. Newly created receptors are appended to the receptor list corresponding to their event ID.

To identify events, it would be more efficient to use a binary search tree. Since the number of possible events is usually relatively low, it is possible to use the list for this task without loosing to much performance.

4.4.3 Activity table

The most active receptors have to be known by the system. First, the most active receptor must be known for the acquisition process, and to identify the most active wave, i.e. the best hypothesis, in the system. Second, the most significant activity waves have to be known to be continued and damped by the wave damping task. For efficiency reasons, only the n most active waves are considered (in the experiments, n was set to 5). The activity table is implemented as an array with n elements. This array is updated every time an event arrives, and is filled with the most active receptors, if their activity level is above the low activity threshold α_{low}. If there are less than n receptors with a significant activity level, the rest of the array is empty.

4.4.4 Concurrent activity table

The concurrent activity table is a list with an entry for each involved external sensor modality. A list entry is called "Modality descriptor", and contains the unique modality ID, a list with all local copies of remote receptors, and a activity cache list with recently active receptors of this modality.

Whenever a message arrives, the Communication Task selects the corresponding modality descriptor, updates all local copies of remote receptors in this descriptor, and updates the activity cache list. The activity cache list is read by the main mapping task, for the creation of cross edges. Data consistency with concurrent access is provided by the protected list structure, as described before.

4.5 Communication layout for distributed computing

The most important required feature of the communication system is a possibility to broadcast a message to all other sensor partitions. The only messages sent by a partition are messages containing its most active receptors, and are adressed to all other partitions. It would be possible to use mechanisms like ethernet broadcast to address all machines in the local network. This is not very useful, since many computers not belonging to the mapping system would be affected, while it would not be possible to connect computers from different subnets to one mapping system.

A better approach is to create a virtual peer-to-peer network, where every participant knows all other participants, and newcomers have to register at the existing communication system. The registration process works as follows: A newcomer (client) sends a registration request, containing its own address, to a participant (server) which already knows all other participants of the system. The server assigns a unique ID, sends it back to the client, and updates its own participant list with the new member's ID and address.

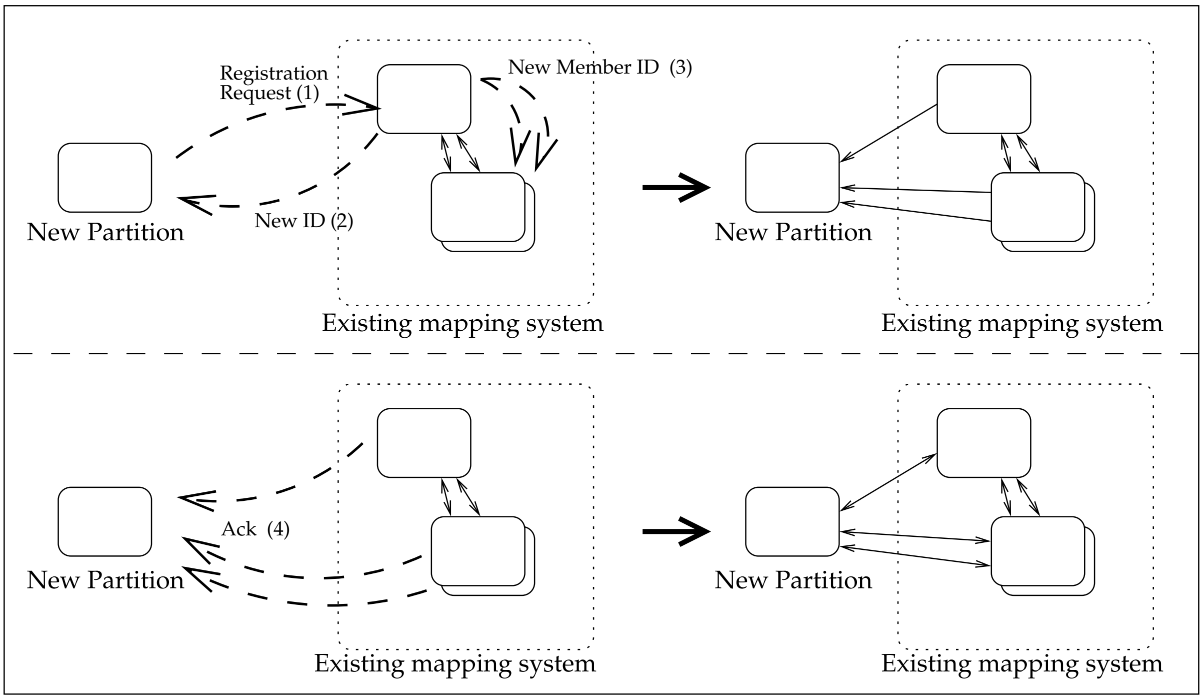

Figure 4.1: The registration of a new member. Thin solid arrows indicate knowledge relations, thick dashed arrows represent messages.

Now the server sends a “New Member” message to all other participants. Those update their local participant lists with the ID and address of the new member, and send an “Acknowledge” message to the new member. The server also sends an “Acknowlegde” message. The new member now receives “Acknowlegde” messages from all other participants, containing their IDs and addresses, and builds up its own local participant list.

After the registration process, all participants of the system hold a complete list of all other participants, which enables them to broadcast messages. Once running, the system doesn’t rely on a central server, which improves the robustness.

Two different communication methods have been implemented.

4.5.1 Ada RPC and distributed object dispatching

This implementation uses the Distributed Systems Annex of Ada95, in the GLADE implementation. It offers Remote Procedure Calls (RPC), and is fairly independent of the underlying network protocol.

For addressing of the symmetric partitions, distributed object dispatching is used. A common, abstract ID type (“AbstractID”) and an abstract message passing “SendEvent” procedure is defined. This procedure expects three parameters: a class wide access to an object of type “AbstractID”, the message to be delivered, and the numerical ID of the sender. The class wide access type for “AbstractID” is declared as a “Remote Call Interface”.

Every partition defines a type which inherits from “AbstractID”, and overwrites “SendEvent”, using the local inherited type. The complete ID of a partition consists of a numerical ID which was assigned during the registration process, and an ID object of the local derivate of “AbstractID”. To pass a message, an access to the ID object of the receiver is required. A call of SendEvent with the receiver’s ID object is automatically dispatched to the partition holding the implementation of that procedure with the respective ID object access.

For registration, a partition server is required. This server offers a remote call interface with three procedures.

The “Register” procedure has an input parameter, which is the access type for the “AbstractID” derivate of the new member, and an output parameter delivering the numerical ID to the caller. The registration process was described earlier.

The “Broadcast” procedure passes a message to all clients known to the partition server. The numerical ID of the sender has to be provided.

The “AddressResolution” procedure expects a numerical ID as input, and returns the ID object derived from “AbstractID” belonging to the member with the respective numerical ID.

Every partition builds up a local list of all participants, containing their numerical ID and an access to their ID object. With every message, the numerical ID of the sender is delivered. If the corresponding ID object is unknown, it can be obtained from the partition server, using the "AddressResolution" procedure. After all partitions are registered, the partition server is not needed any more. Every partition is able to broadcast messages to all other members.

4.5.2 TCP/IP and UDP/IP communication

Another communication system using Unix Sockets has been implemented. The basic structure is identical, but the central partition server is not needed any more. A new member can register at any participant of the mapping system, making it an absolutely symmetrical distributed system.

The complete ID of a partition now contains the IP address plus port number, and a numerical ID. The registration process is identical to earlier descriptions, with the difference that any of the symmetrical partitions can be the registration server now.

The implementation using TCP/IP offers a more reliable communication, but caused some problems concerning realtime constraints. A message containing the most active receptors which cannot be sent due to poor network conditions, must not block the system. This applies also to the case when the receiver accepted a TCP connection, but the actual message never arrives. In both cases, a time-out must be implemented, and the communication must be aborted. A lost message just slows down the sensor fusion process. This is not problematical, but a delayed execution of input data with possible loss of samples is. In some cases, an abortion of the Linux/Unix Socket system call lead to an undefined state of the communication interface on the operation system side. This caused a loss of the SIGIO signal, probably due to processor overload, which affected the communication with the SICK Laser Sensor over the serial port.

The UDP implementation doesn't suffer these problems, since it is not connection-oriented, and both the "send" and "receive" Linux/Unix system call are non-blocking. For realtime applications, a UDP communication system is often more appropriate. For this system, packet loss has no severe consequences and can be ignored, since every partition is fully operative without any communication. Of course, sensor fusion is slowed down by large scale packet loss, but this only has an impact on performance, but not on principal functionality. UDP communication is the appropriate choice for this system, and proved functionality and efficiency in the experiments.

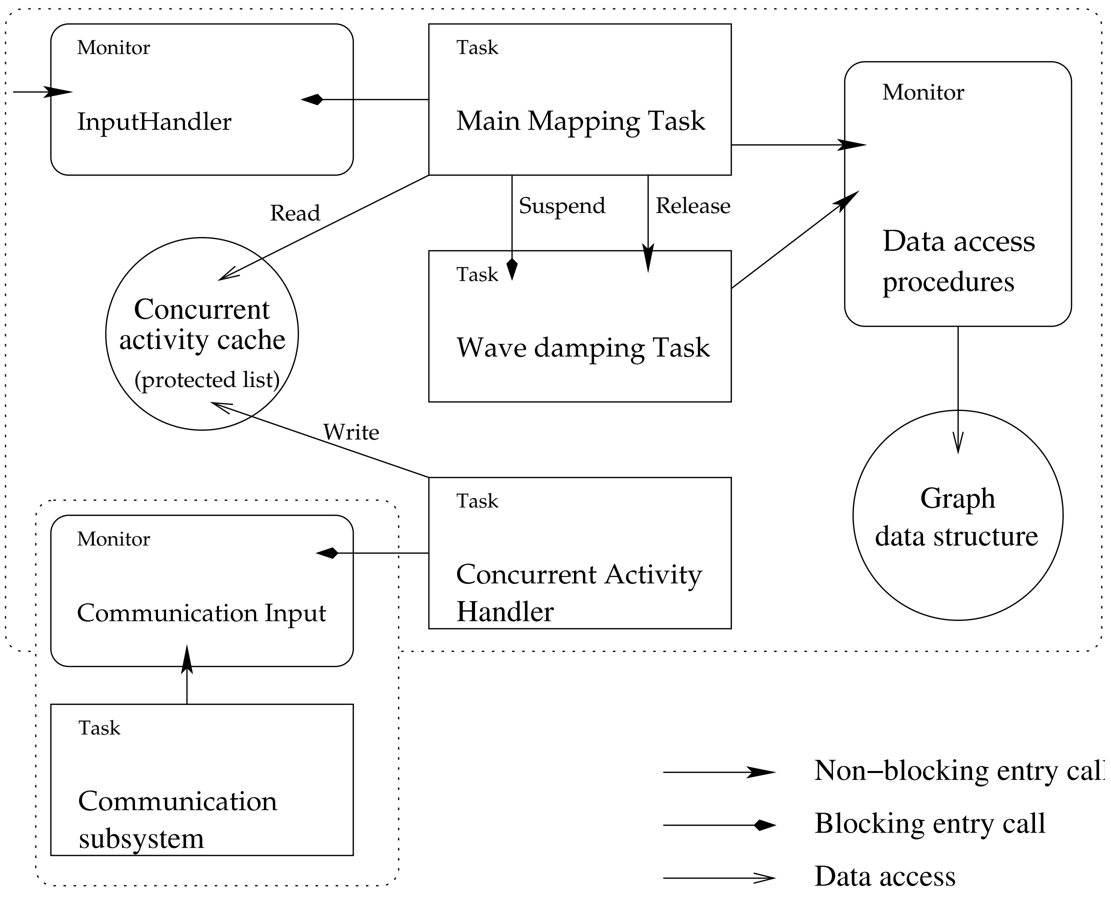

Figure 4.2: Illustration of the most important parts of the system and their dependencies

4.6 Dependencies

An important question in multitasking systems is if deadlocks can occur. Deadlocks can be caused by concurrent resource dependencies, or circular task wait conditions. All data access procedures have been designed non-blocking and atomic. Where the protected list is used, this is already provided by the non-blocking atomic list operations. More complex operations have been declared in a separate monitor, to guarantee atomicity within the system. The result is that all concurrent data accesses on common data structures are serialised. This makes locking mechanisms unnecessary, and avoids deadlocks. It would be possible to allow concurrent read access in some cases. But since only three tasks are accessing the data, and read conflicts hardly occur, this would not increase performance a lot.

There are dependencies between the tasks and monitors of the system (see fig. 4.2. The main mapping task and the wave damping task have to run in mutual exclusion. Before starting a calculation cycle, the main mapping task blocks itself while trying to suspend the wave damping task, until the latter has finished its cycle. After this, the main mapping task is released, and the wave damping task remains blocked. When the main mapping task has finished its cycle, it resumes the wave damping task again. This guarantees mutual exclusion. Deadlocks can't occur here, since the main mapping task is only blocked while calling the suspend entry, and the suspend entry is only temporarily blocking. The wave damping task does not depend on the main mapping task, so there is no circular dependency.

All other tasks only block themselves at monitors, waiting for new input data. The monitors can't lock, and as long as new data arrives, the system is alive.

Chapter 5

Experiments

The expected functionality of the proposed system is to recognise and identify repeating temporal patterns or rhythms in the input data streams. To evaluate the capabilities of the system, several experiments have been accomplished, using simulated and real sensor data.

The discrete multiple sensor system has been implemented with the programming language Ada95, including the realtime annex. It has been compiled with the gnat compiler for 32 bit Intel 80x86-compatible linux platforms. The final version used for the experiments communicates over the UDP/IP protocol.

5.1 Measurements and criteria

To measure the success of the method, respective the quality of the mapping and recognition process, the following measurement shall be introduced. The system has to make one guess, which receptor will be the most active in the close future, everytime an event arrives. This guess is determined by simulating a perfectly timed activation of the receptors on each outgoing edge of the currently most active receptor. For this simulation, all information including cross edge activity is used. The receptor which can achieve the highest activity level in this simulation is guessed to be most active in the close future. When the available information from cross edges changes, the guess may be modified, but only before the next event occured. A correct guess is rewarded with the value 1. A guess expires, if the time span represented by the connecting edges from the currently most active receptor have expired. An expired guess results in a 0. A sliding mean value $P(t)$ (prediction success) represents the outcome of the last guesses.

Predicting the next most active receptor is a very strong criteria. The activation of a certain receptor not only depends on the occurence of the right event in approximately the right time, but also in the history of events, since the activity level of the input edges is relevant. A correct guess therefore expresses a big

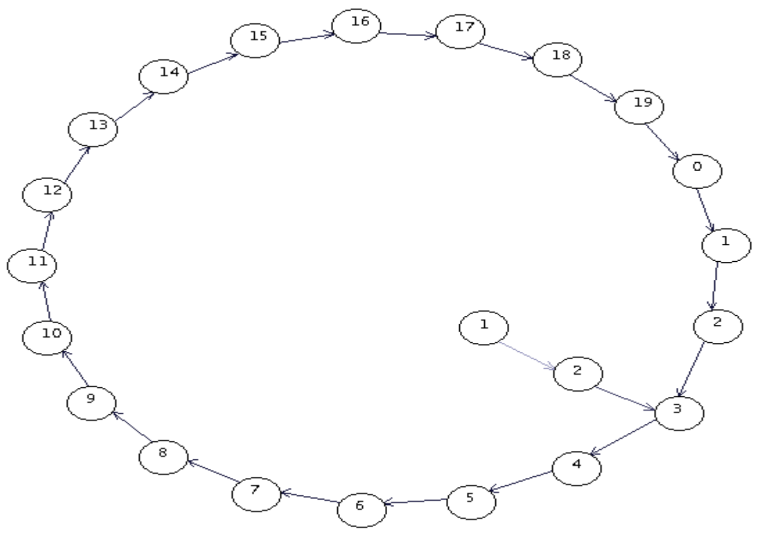

Figure 5.1: Internal representation of the input data from the second experiment with perfectly reproducible simulated data

certainty of the system about the current and future input data, and is a good quality measure of the current most probable hypothesis of the system. The noise level of $P(t)$ depends on the number of events and the variance of time spans between events. Assuming that events arrive with a constant frequency, and all transitions have the same probability (white noise), the expectation value of $P(t)$ is below $\frac{1}{|E|}$, with $|E|$ being the number of possible events. For varying time intervals between events, the expectation value of $P(t)$ is lower. The actual expectation value depends on the timing variety of events, which is not restricted and thus hard to determine. It is also dependent on system parameters such as the initial temporal tolerance σ_E, the amplification parameters, and the adaptation parameters.

5.2 Simulated data experiments

Real sensor data from real world experiments has a complexity that makes it difficult to analyse the behaviour of a system reacting to this input. To get a better understanding of how the system works, it is useful to construct some artificial scenarios which aim for a certain behaviour of the system, and to compare the system's reaction to the expectations. Two basic simulated experiments have been chosen. The first experiment shows the functionality for one sensor input. The second experiment analyses the functionality and the impact of sensor fusion.

5.2.1 Clean and noisy input comparison

To test the system's basic functionality, the signal to noise distance of the prediction success measure $P(t)$ is determined by feeding the mapping system with a sequence of evenly distributed random events, and with a sequence of events consisting of one repeating subsequence. To simplify the experiments, the time interval between events is constant. Any real world experiment will add complexity due to the varying time base of event sequences.

In the first experiment, every 0.2 seconds one event out of 20 unique events is randomly selected (using the Ada random numbers package), and fed into the mapping system. It is expected that the system will reach a low activity level, since no sequence is repeated accurately in this experiment. The $P(t)$ measurement should reach the statistical prediction average of 1/20.

In the second experiment a sequence of 20 unique events is repeatingly fed into the system, again with 0.2 seconds delay between two events. After a few repeated observations of this subsequence, it should be internally represented and recognised by the system, resulting in a high activity level. The $P(t)$ measurement should reach its maximum value, since there is no ambiguity, and all predictions should be correct.

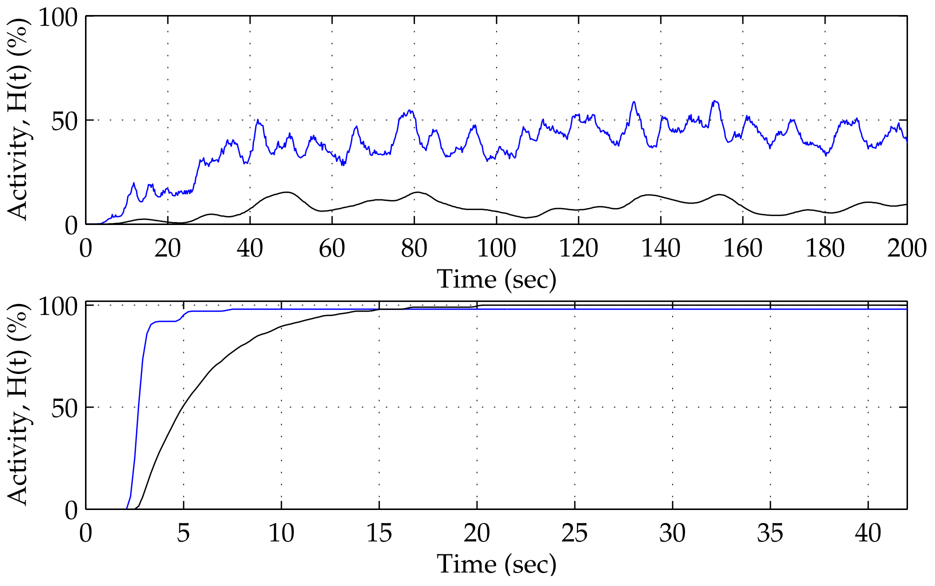

Figure 5.2: Activation (blue, upper curve) and $P(t)$ (black) for random input data (upper plot) and for absolutely reproducible input data (lower plot)

The expected behaviour can clearly be seen in figure 5.2. It takes approximately 50 seconds or 250 events for the system to adapt to most of the occuring single-step transitions. This results in a visible, but still low activity level. The activity level ususally remains below the recognition threshold $\alpha_{high} = 0.5$. Still there are no accurate predictions possible, which is represented in a low value for $P(t)$. The prediction success rate is approximately the statistical average of 1/20.

In the second experiment, the same sequence repeats every 4 seconds. It can be seen how the activity level quickly reaches the maximum, when the sequence is observed the second time. The value $P(t)$ grows exponentially, since all predictions are correct from that moment.

5.2.2 Sensor fusion in simulation

This experiment is to investigate the functionality of sensor fusion. A sequence of 10 events, containing one bifurcation point, is used as input. At the bifurcation, each path has a 50% probability, and is randomly chosen. The experiment setup consists of two mapping systems, which communicate over UDP. The input sequence is repeatedly fed into the first mapping system. At the bifurcation point, a random decision is made. The second mapping system receives exactly the same input sequence, with the same decisions made, but with a short delay.

Without sensor fusion, both mapping systems should achieve a prediction success rate lower than 100%, since the bifurcation decision cannot be predicted. With sensor fusion, the mapping system with the delayed input should be able to use the information of the other mapping system to predict the bifurcation decision correctly.

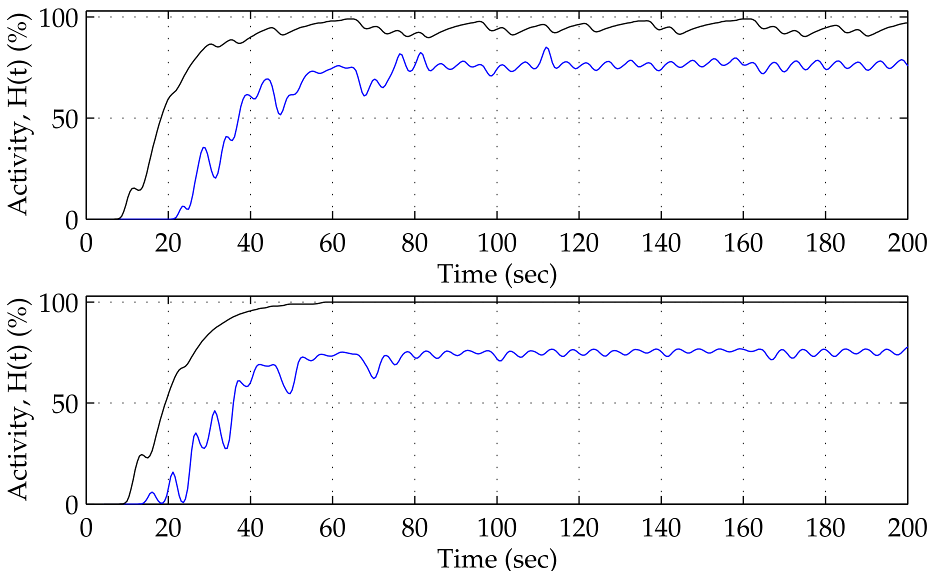

Figure 5.3: Activation (blue, upper curve) and $P(t)$ (black) for the preceding (upper plot) and succeeding (lower plot) mapping system

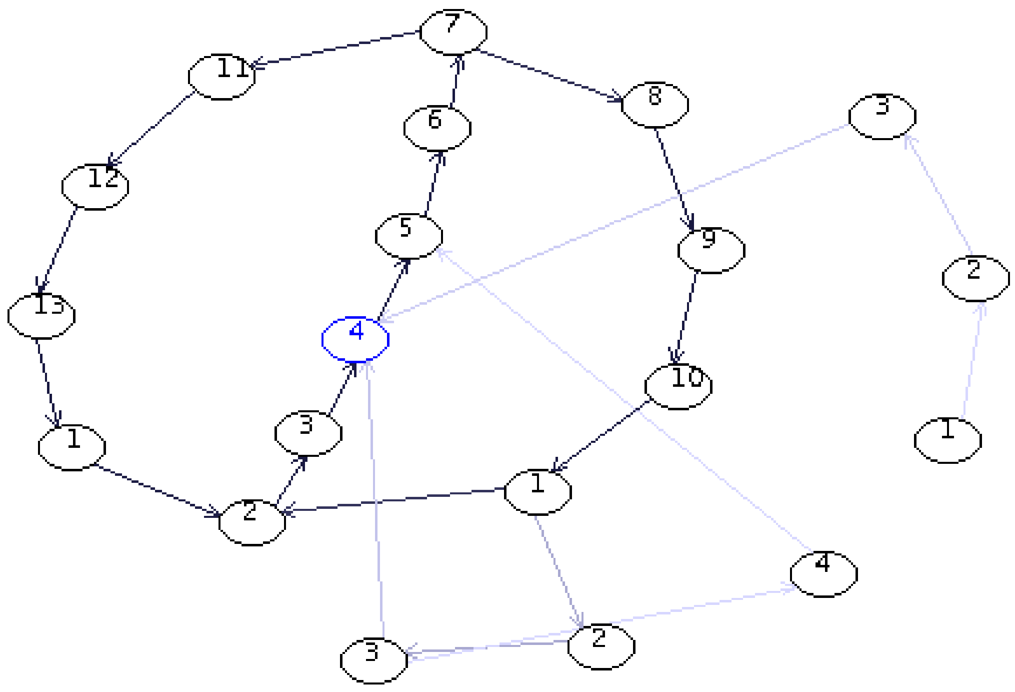

Figure 5.4: The internal graph structure of one of the mapping systems in the simulated sensor fusion experiment.

It can easily be seen in the plots (fig. 5.3) how the succeeding mapping system (lower plot) produces 100% correct predictions, after the sensor fusion process has stabilised. This shows that the described method for fusing mapping systems with cross edges enables modalities to consider information from other modalities to improve recognition and predictability of input data.

Experiments with different delays for the input of the second system produced similar results, except in two cases: If both systems receive the same input exactly synchronized, neither of the modalities shows improvements. This is clear, since no further information about the further path at the bifurcation point is available, when the prediction has to be made. The other case is when the delay is bigger than the system's time window (the maximal size of R_{lma}, see 3.1.1), so that no temporal correlations can be observed any more. In both cases, both systems achieve the same performance as if working alone. Here another feature of the system appears: Correlations across sensor modalities are used where they exist, and performance is improved. Where no reproducible correlations exist, external influence is ignored.

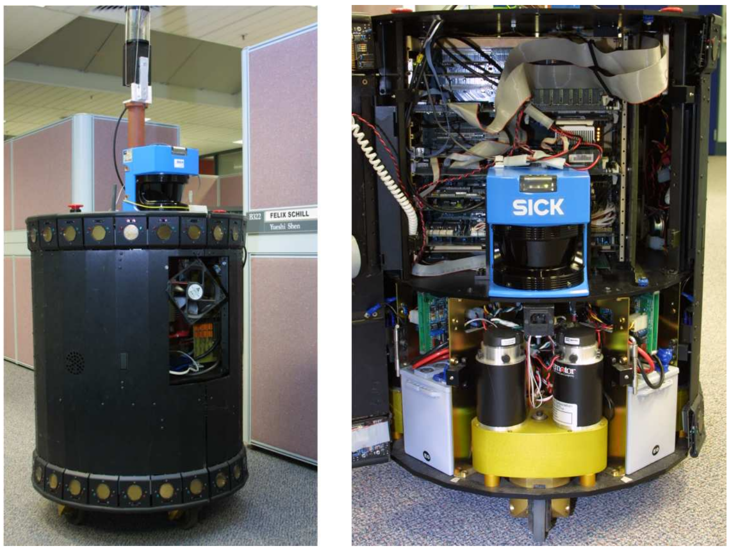

Figure 5.5: The Nomad XR4000 mobile robot

5.3 Real world experiments

It has been shown that the mapping system shows stable behaviour with noise and perfect input data, and that sensor fusion works in simulation. Input data from real sensors is always interfered with noise. The complexity of reality always leads to ambiguities, wrong measurements, and unpredictable events. This section examines the behaviour and stability of the mapping system under real world conditions.

To obtain event sequences, raw sensor data is clustered, using the clustering system described in 2.5.2. The events resulting from cluster transitions are used as input for the dynamical mapping system (see also section 2.5.2).

5.3.1 General experiment setup

A landbound autonomous mobile robot has been chosen for the following experiments. While the robot drives around using a reflexive exploration strategy, the mapping system processes incoming sensor data, trying to recognise known patterns. As in previous experiments, the activity level of the most active receptor and the quality measure $P(t)$ are recorded and evaluated. The exploration module is independent from the mapping system in these experiments, except that both are using input data from the same laser sensor.

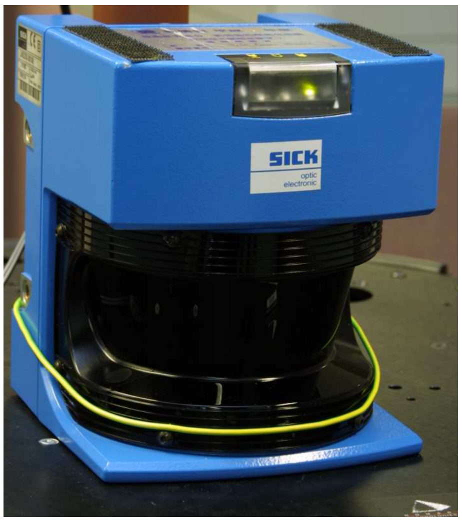

Figure 5.6: SICK laser sensor

The Nomad XR4000 mobile robot

The Nomad XR4000 is a holonomic mobile platform. Cylindrical in shape, it has a radius of approximately 30 cm, and the main body is about one meter high. It is equipped with an internal SICK Laser Range Sensor, 48 infrared and sonar proximity sensors, 48 collision sensors (bumpers), two identical on board Pentium III computers with 633Mhz, and batteries allowing up to 6 hours of operation without recharging.

The two internal computers are called “Jekyl” and “Hyde”. All built-in sensors and the motor controllers are attached to “Hyde”. Both conputers are interconnected with ethernet, and with a radio ethernet adapter, enabling the robot to communicate with the internet. The maximum speed of the platform is 1.5 m/sec in any direction, which is reached after one second at full acceleration. The maximal angular velocity is 90 degrees per second.

On top of the platform, more sensors can be and have been mounted, such as a second SICK laser sensor, a panoramic camera, or a stereo vision system. For the experiments, only the two laser sensors and the sonar sensor have been used.

SICK LMS400 Laser Sensor

The SICK LMS400 measures the distance to the next obstacle, using the time of flight of an infrared laser beam. The range is 180 degrees angular and up to 32 meters linear, at an angular resolution of 0.5 degrees. A full scan consists of 361 distance measures. The scan rate in the configuration used for the experiments is 5 scans per second. Measurements are very reliable for almost any kind of obstacle

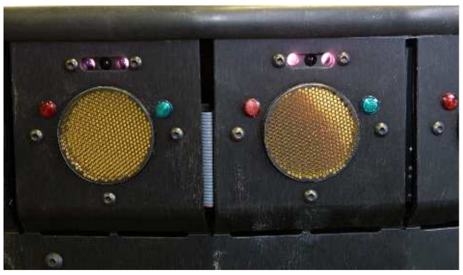

Figure 5.7: Sonar sensors

surface, including glass. A quality information for every single measurement is available.

Two SICK laser sensors have been used. One is mounted inside the robot, scanning forward-looking parallel to the ground in a height of 50 cm. The second laser is mounted on top of the platform, scanning backward-looking 1 meter above the ground. The first sensor is connected to the primary computer, called "Hyde". The second laser is attached to "Jekyl", the second computer onboard the robot.

Polaroid Sonar Sensor

The Nomad is equipped with 48 Polaroid sonar sensors, which are located at the upper and lower rim of the main body. The sonar sensors are time-of-flight distance sensors with a measuring range of 20cm to 5-6 meters. A single pulse is emitted by an ultrasonic transmitter; the echo is received by the same transmitter.

Measurements are error-prone due to multiple reflections, interference, and poor echo signal quality on many obstacle surfaces. To avoid interference between the sensors, neighboured sensors emit pulses one after another, with delays long enough that all echos of the previous pulse have disappeared. The overall scan frequency is approximately 1 Hz.

Environmental setup

All experiments were carried out on corridors in an office environment. The available space was limited to an L-shaped section. Observable features were open doors, poles, and intersections. The uniform appearance of the environment makes localisation more difficult, since different places may be hard to distinguish in the sensor data. Different locations often resulted in the same cluster categorization after the online clustering process. This emphasizes the importance of the context of observations.

Since the experimental environment was an office area, interferences of the robot with people walking around in the building weren't always avoidable. A person in sensor range not only influences the observed data, but can also influence the robot's behaviour (collision avoidance, path fluctuations). While partial changes in range scans caused by dynamical objects usually have a small or no impact, a changed behaviour affects the spatio-temporal flow of events. This can affect the performance of the temporal mapping system. The mapping system was designed to ignore small changes and noise. Bigger variations of input data or even different paths are learned during observation. Of course, such variations can't be known in advance, and some variations are not likely to happen again in the same manner. In those cases, the mapping performance can decrease.

The floor of the office area is covered with carpet. Experiments proved odometry to be useless due to drift effects. After two passes through the L-shaped section, the internal coordinates already drifted between one and two meters. It was not possible to use odometry for evaluation purposes. Instead, landmarks were manually entered.

5.3.2 Exploration strategy

It is vital for the mapping technique that the robot moves around in a more or less predictable and reproducible manner. That means that the robot behaves similar in a similar situation. Of course, an exploration strategy should also avoid collisions, and should be able to cope with dynamical objects.

An exploration strategy based on reflexive motion patterns has been implemented. There is no connection to the mapping process; exploration is independent from the mapping status.

For collision-avoiding exploration, the internal SICK laser sensor is used. A simulated 360 degree map is calculated while moving, using odometrical information. All measurements in this map are extended by the robot radius plus a safety distance, so that the resulting 360 degree map represents the distances the robot can go without a collision in the corresponding direction.

Now the robot simply turns into the direction with the maximal possible distance to go. If this direction offers only a distance too small to be interesting (for the experiments, 0.75 meters were found to be appropriate), the robot turns around until a better direction can be seen. A forward movement is combined with the rotational movement at the same time, where the forward speed is controlled by the minimal distance to the next obstacle in sight.

This results in a smooth movement along the corridors of the laboratory. When reaching a dead end, the robot turns around. The approach is stateless and reflexive, nevertheless stable and reliable, even in dynamical environments. Since the forward movement is controlled by the distance to the next obstacle, collisions with static objects are not possible. The robot also choses a direction in which it can go furthest, and thus is keeping away from potential obstacles. When an obstacle is approached, the robot will slow down and try to find a way around it. If there is not enough space to get around, it will turn around and chose another direction. The paths through the environment will never be absolutely identical. When visiting a place again, the robot behaves in a reproducible way, which is sufficient to get similar temporal patterns in sensor data.

5.4 Software setup and constants

First of all, an interface to the robot's controller software had to be created. For the SICK Laser Sensor, a driver has been developed in Ada95. The sonar sensor, the motor controller and all other sensors where accessed using the Nomad C-library delivered with the robot. Therefore, an Ada interface to the C library was developed, that made it possible to access all needed controller functions by native Ada subroutines. The Ada interface also converts all measures into metrical measures, and performs some basic range checks, to simplify access.

For every sensor, the raw data was preprocessed by the online clustering system. The resulting event stream was the input for one partition of the temporal mapping system. A partition was executed on the computer to which the corresponding sensor was connected to. In the experiments with sensor fusion, the partitions communicated over UDP sockets. Every partition created verbose logfiles, containing all relevant information including timestamps, for evaluation purposes.

The clustering system was set up individually for each sensor type. The sensibility was chosen, that about 50 clusters per sensor were created during the experiments. The configurations of the mapping systems for each sensor were identical. The following settings were used:

Parameter	**Value**	**Description**
α_{high}	0.5	High activity recognition threshold
c_a	5.0	Amplification factor
c_{damp}	0.3	Damping factor
ε	0.01	Self-activation term
c_R	0.2	Firing energy recharging rate
λ_i	0.8	Initial learning rate
c_λ	0.3	Learnrate reduction factor
c_d	0.03	Inactive edge decay factor
σ_i/δ_i	0.5	Initial temporal precision
t_{pmax}	0.1	maximal temporal precision (minimal σ/δ)
c_t	0.4	time warp adaptation rate

The number of transmitted active receptors for sensor fusion was limited to 5. A message sent to other partitions via UDP fitted into one UDP packet.

The constants and parameters have been chosen according to the results of many preliminary experiments. Some parameters are not critical, and would be applicaple to many other situations. But especially the sensibility of the clustering system, the amplification factor and the self-activation term should be adapted to the target environment and the sensors used.

5.5 Results

Several experiments with the described robot were carried out. Using the exploration strategy described before, one complete pass through the L-shaped environment took about 120 seconds.

5.5.1 Experiment 1: one internal laser sensor only

This experiment is to show the principal functionality of the system for one sensor modality. As in all following experiments, the exploration strategy described above is used. The calculated 360 degree map calculated by the exploration module is used as input for the dynamical system.

As visible in fig. 5.8, the activity level of the system increases after the second or third pass through the environment. This means the system already recognizes sequences of the input data. After the fifth pass, the mapping system for the forward looking internal sensor (“Hyde”) is able to reach a prediction success level of about 40 percent. This value increases up to 50 percent, while the experiment continues. The second mapping system for the backward looking laser (“Jekyl”) needs a bit longer to stabilise, due to higher complexity of the input data, but after 10-12 passes, it stabilises with a prediction measure $P(t)$ of 45-50 percent.

The prediction success of 50 percent appears to be low. In fact, it is a good success, considering that temporal and causal information has to be guessed correctly. Second, there are multiple parallel hypotheses in the system, which are roughly equivalent and just represent variations of the input due to noise. Since only one guess per incoming event can be given, it is not possible to predict everything. The activity level soon reaches a level above 0.5, which means that the system recognizes the input.

5.5.2 Experiment 2: two internal laser sensors and one sonar sensor

This experiment investigates the improvement of recognition when using several sensors instead of only one.

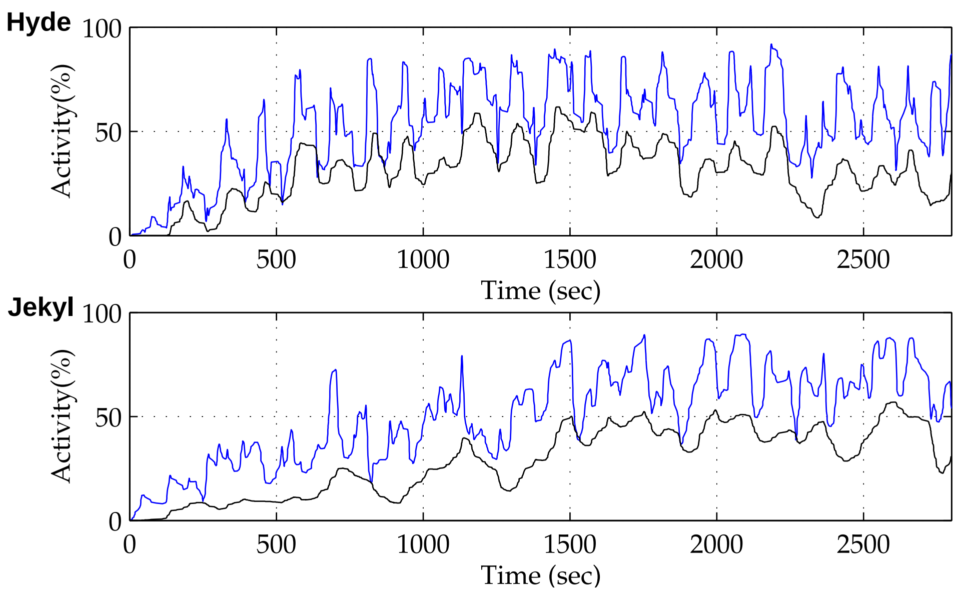

Figure 5.8: Measurements for standalone laser sensors (no fusion). The blue curve shows the receptor activity, the black curve is the percentage of correct predictions P(t).

The second laser sensor, mounted on top of the platform looking backwards, has been used for this experiment. Due to a different height above the ground (1 meter instead of 50 cm), this laser observes a different part of the environment.

One mapping system uses the 360 degree map as input, which is calculated using data of the forward looking laser sensor, as in experiment 1. The second mapping system uses raw data from the backward looking laser sensor. The third mapping system is connected to the sonar sensor. All mapping systems communicate over UDP/IP, to achieve sensor fusion as described in chapter 3.

The results (figure 5.10) show that the two laser sensors obviously cannot profit from sensor fusion in this experiment. Only the sonar sensor can significantly improve its (overall poor) performance. It is not absolutely clear why the results for the laser sensors are not improved by sensor fusion. One explanation is that the intrinsic noise in the experiment setup does not allow a higher prediction success rate than already achieved in the single sensor case. All sensors provide data with similar characteristics, and in consequence all could suffer from the same kind of intrinsic noise. If the observed noise does not come from the measurements, but is already present (and the same) in all measurands, sensor fusion can't have the desired effect. Further investigations on that topic in experiment 4 support this theory.

The next experiments use a different sensor setup, where the different sensor modalities should have different characteristics towards the measurand and the

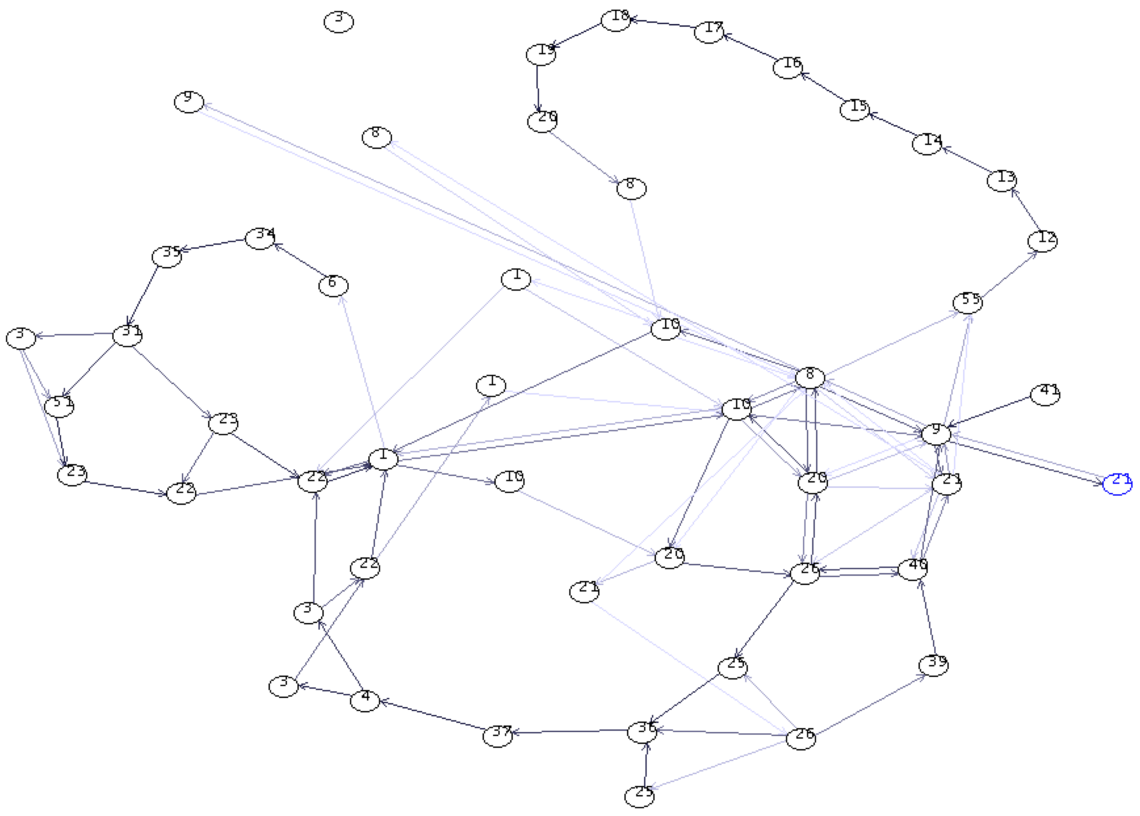

Figure 5.9: Established paths in the internal graph structure of the single laser sensor experiment (unused receptors and edges have been removed for better clarity)

kind of noise.

5.5.3 Experiment 3: internal laser sensor, external laser sensor

The robot is in the same configuration as in experiment 1. Again, the 360 degree map is used for mapping. Additionally, a stationary laser sensor observes the robot driving around. The measurements of this external sensor are used as input for a second mapping system. Again, the two mapping systems communicate over UDP/IP.

The external sensor cannot see the robot at all times, and thus doesn't contribute everywhere. It is expected, that the activity and the $P(t)$ measurement are higher when the robot is in sight of the external sensor. When the robot is out of sight of the external sensor, results are expected to be comparable to the results of experiment 1. This experiment also demonstrates the flexibility of the system.

The expected effect can clearly be seen in the plots in figure 5.11. The thick blue

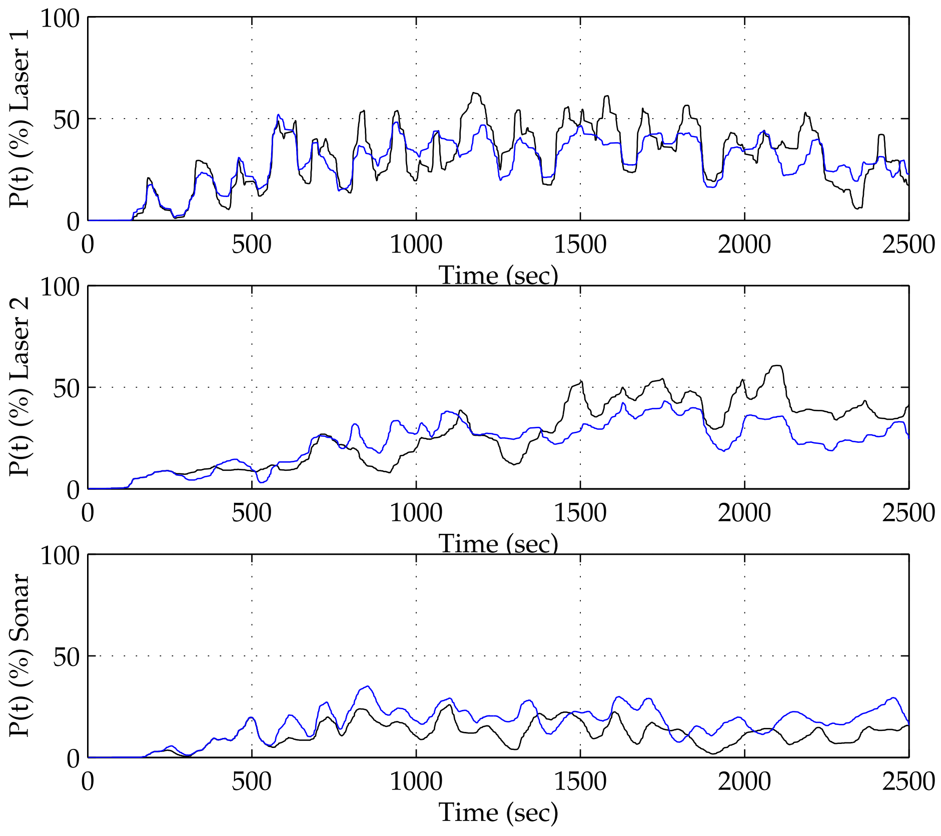

Figure 5.10: Results of the experiment with all sensors. The black curve shows the results without sensor fusion, the blue curve with sensor fusion.

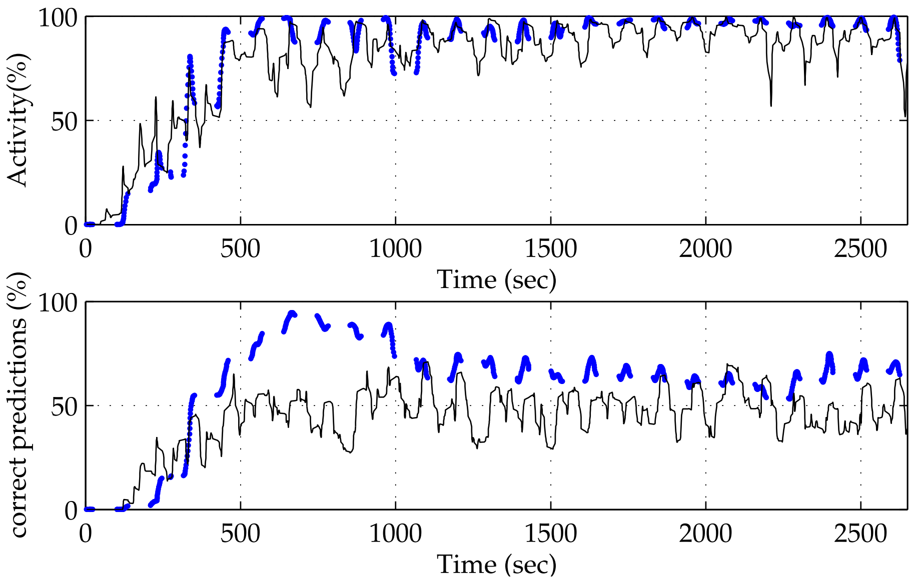

Figure 5.11: Experiment with an internal and an external laser sensor. The black curve represents the internal sensor, the blue curve shows the external sensor measurements.

plot (belonging to the stationary laser sensor) has gaps, whenever the robot is out of sight. The activity level and the prediction measure of the internal sensor are clearly higher at times when the external sensor observes the robot and delivers data. The peaks in the thin black plot correlate well with the fragments of the thick blue plot.

The same experiment has been carried out without sensor fusion, using exactly the same input data. The results are compared in figure 5.12. It can be seen that sensor fusion improves the results shortly after the startup phase. After about ten passes, the systems not using sensor fusion can catch up and achieve approximately the same performance. This indicates that the saturation of the prediction measurement at about 50 percent has its foundations in the experiment setup. Obviously there is a certain amount of noise, which is common in all sensor modalities, and a prediction success rate of 50 percent is close to the possible maximum in this special experiment setup. This noise probably comes from the variations in paths chosen by the exploration strategy. Another source could be clusters which are very similar. If the clustering system oscillates between two similar clusters, the event chain is not absolutely predictable.

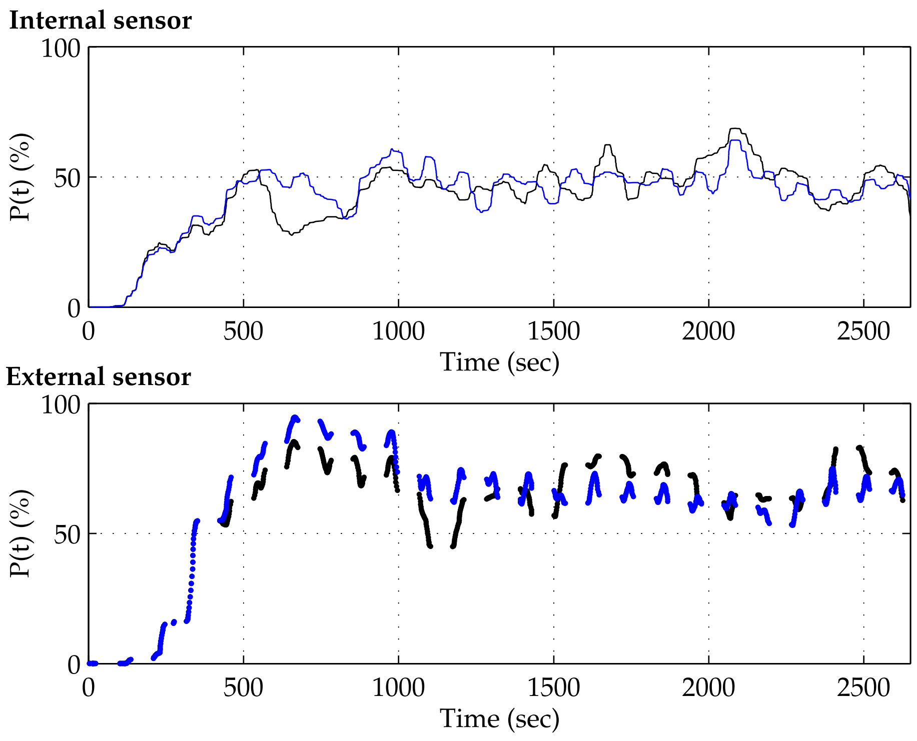

Figure 5.12: Experiment with an internal and an external laser sensor. The black curve shows the results without sensor fusion, the blue curve with sensor fusion.

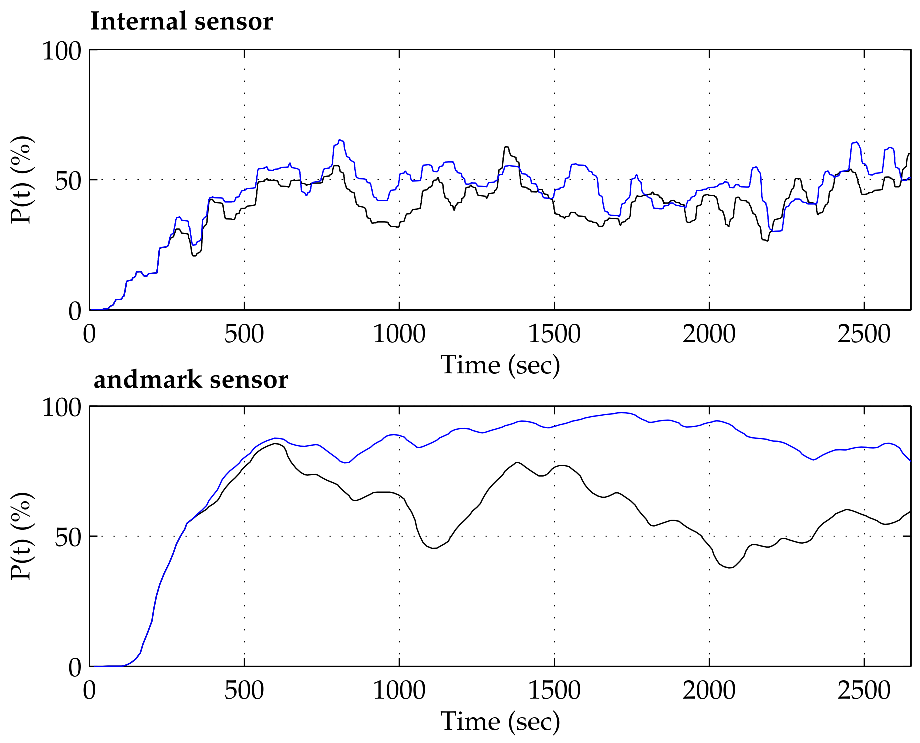

Figure 5.13: Experiment with an internal laser sensor and manual landmark input. The black curve shows the results without sensor fusion, the blue curve with sensor fusion.

5.5.4 Experiment 4: laser range finder and manual landmark input

In this experiment, the robot again is in the same configuration as in experiment 1. A second (external) sensor modality is given by a human entering landmark identifiers into a computer, when the robot reaches the corresponding position.

This experiments targets behaviour of the system when dealing with extremely different timing behaviours of the sensors. The laser sensor delivers data 5 times per second, and approximately every 1-2 seconds an event is detected. The average time between two landmarks is 8-12 seconds.

The results of this experiments are shown in figure 5.13. Obviously the landmark sensor modality is very reliable. There is only some noise in the time domain (depending on the speed of the robot). The influence of this reliable modality clearly improves recognition results for the internal laser modality. This again shows the functionality and flexibility of the sensor fusion method, even when dealing with extremely different data rates and sensor characteristics.

On the other hand, the laser sensor improves the results of the landmark sensor, too. Without sensor fusion, the landmark sensor is sensible to temporal noise. When using information of the laser sensor modality, the prediction success ratio increases, since the extra information implicitely contains the speed of the robot in the current section. This makes better predictions of the expected arrival of an event possible.

The plots of this experiment clearly show the effects of sensor fusion. In the first two passes through the environment, sensor fusion has no effect, since both systems have to establish some structure first. But already in the third pass (after about 250 seconds), slight improvements by sensor fusion are visible. After 5 passes, intermodal correlations have stabilised and clearly improve recognition and system stability. The prediction success ratio of the laser sensor mapping system still saturates at about 50 percent. The correct position is known by the landmark modality, but this cannot improve the predictability of the laser sensor modality. This indicates that 50 percent prediction success is close to the maximum which can be reached using this sensor with the current clustering method.

5.5.5 Experiment 5: Attaching and comparing landmark information

This last experiment investigates the correlation of internally recognized positions (receptors) with locations in real space. Since the internal topological map is not easily interpreted by human operators, it is important to introduce landmarks defined by the operator into the internal world representation.

This time the manually entered landmark information is not a sensor modality in the system. Now, when a receptor is created, it stores the last landmark identifier which was entered. As the experiment continues, the landmark information stored in the currently most active receptor (the position which the system recognized) is compared to the current true location (the last manually entered landmark). The success rate is shown in figure 5.14.

After stabilisation, the system correctly identifies the current landmark for 70-80% of all incoming events. This result is very good, considering that no informations about the sensor characteristics have been used, and the system already stabilises after 4-5 passes through the environment.

It can be observed that wrong identifications usually have significantly lower activity levels, but this is not considered in the measurements. Another source of identification errors is that receptors cannot change their allocation to a landmark, even if the topological structure changes in the stabilisation phase. For a real application, the results could be improved by weighting landmark identifications with the corresponding activity level and prediction success, and adapting the receptor-landmark relations.

This experiment impressively shows how landmark information can be introduced into the system, and that the internal world representation correlates with

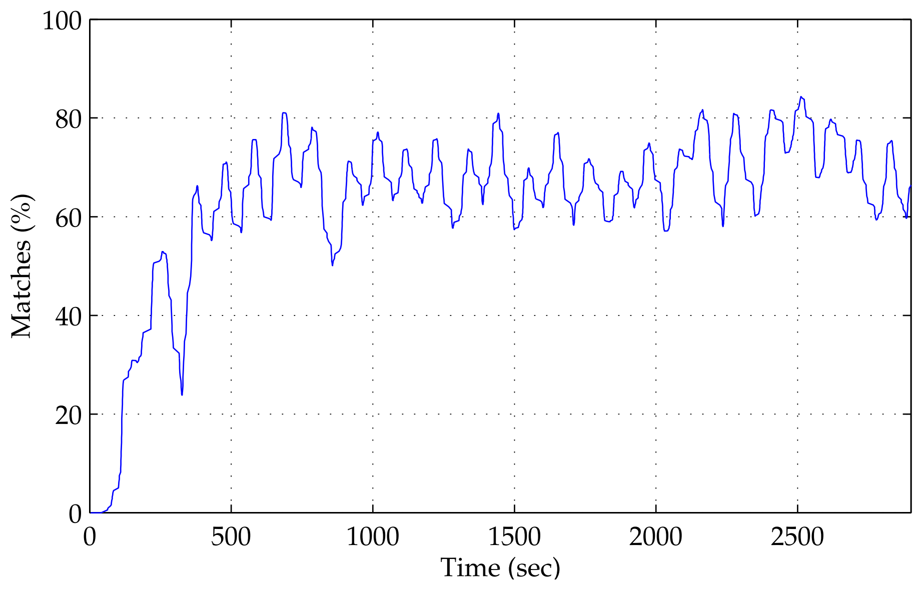

Figure 5.14: Landmark identification success in percent.

real world locations. External landmark information is used in this experiment when receptors are created. But landmark information could be introduced at any time by modifying most active receptors. This is relevant for practical applications, where a human operator has to provide location identifiers to create a communication basis to the navigation system. Once familiar with landmark identifiers, the system can translate target specifications into the internal world model, and vice versa. Only after this is possible, a human-machine communication is possible.

Chapter 6

Conclusion

A dynamical system for temporal event mapping has been developed. The system has no rigid time base, and is able to deal with all kinds of input frequencies and rhythm variations. The number of possible events is flexible and can change during runtime. The map size grows dynamically as required, depending on the input. The only limitations are given by the memory size and speed of the computer executing the mapping system. The computational effort for input recognition is low, the system is able to work online. It has been experimentally shown that it is possible to use the presented temporal mapping method for simulataneous localisation and mapping.

The aquisition of input sequences is immediate, and the adaptation process is very fast. In simulation, a sequence is recognized already the second time it occurs. In real world experiments with online clustering of raw sensor data, locations are robustly recognized after 4-5 passes through the environment.

An extension has been presented to fuse information from sensors with different characteristics and timing behaviours to improve recognition results. The dynamical system was implemented as a symmetrical distributed application. Functionality and efficiency have been investigated in simulation and in real world experiments. The presented approach overcomes some drawbacks of other methods in terms of flexibility towards range, complexity and temporal characteristics of input data.

The proposed mapping method can be applied to robots in environments which cannot be easily modelled, and where movement in space cannot be measured directly by odometry, such as on submarines and flying or hovering robots. It offers many possibilities to use sensors which could not provide any useful information otherwise, since no explicit feature extraction or sensor data interpretation is required. Examples for a submarine are sensors for water temperature, pressure, electric conductivity, or chemical properties and water quality. Those sensors don't provide great amounts of information about the position in a single measurement. The temporal information in a sequence of those measurements is much greater, as well as the information in correlations of different sensor modalities. There may be situations for a submersible robot, where no other information is available. Using the proposed system, this information can be used for localisation, and can provide essential clues about the robot's position.

There are many other applications for temporal mapping methods like this, and many situations in which temporal information and rhythms in input data are important. This was not discussed in this work, but offers a huge range of further investigations.

6.1 Future work

The basic features and properties of the presented system have been experimentally investigated. Yet there are still many open questions which could not be answered in the limited frame of this work.

There are many applications for the presented temporal mapping system, in robotics but also in other areas. The temporal mapping system can be useful, where an analysis and recognition of timing or rhythms is important. One example is the learning of dynamical trajectories. After some demonstrations of a motion pattern, the system would be able to recognise it, and could try to imitate it. Activity waves could be fed back into the system, so that a short start sequence could trigger the execution of the whole trajectory. Other areas are speech recognition or music analysis.

An important and very complex topic is the thorough theoretical analysis of the presented analog model. This work has its emphasis on the discrete implementation for common digital computers, and experimental examinations. It is not absolutely clear how to analyse a continous, dynamically growing and adapting system of this complexity. A big problem is that the structure and structural changes of the system depend on the input data, which depends on processes in reality. The complexity of reality is infinite, so it is difficult to make valid assumptions and to completely understand the theoretical behaviour of the mapping system in extreme situations.

Another big question is the amount of implicit information contained in temporal patterns, and how it can be measured. Of course, this depends very much on the characteristic of the input streams, the environment, and on the kind of information which should be extracted from the input. Many topics emerge here, i.e. how temporal information can be processed, used and stored, where it makes sense to use it and where it doesn't, and how to learn making this decision.

A practical question is how to use the topological map of the presented system for navigation, and how the method can be combined with other approaches (i.e. cartesian mapping). It is also interesting if and how the similarity measurement of the data clustering method can be automatically adapted to the underlying structure of the input data. This would result in a more appropriate clustering of the input, and in less noise in the event stream.

The vision is an online system based on unsupervised learning, that automatically identifies features in sensor data, finds correlations and temporal patterns between multiple inputs, and is able to adapt to the current set of sensors and the environment. Unsupervised learning usually means finding structures in unknown input data. It always depends on the application, which structures turn out to be useful. The big goal is to find a system, which is able to learn making this decision itself, depending on the task to perform.

Chapter 7

Appendix A: Kurzfassung in deutscher Sprache

7.1 Einleitung

In vielen Fällen ist es schwierig, Sensordaten geeignet zu modellieren. Zum Beispiel werden Ultraschall-Entfernungssensoren auf einem Untersee-Roboter durch unzählige Faktoren wie Wasserdruck, Temperatur oder Salzgehalt beeinflusst. Der hier verfolgte Ansatz umgeht dieses Problem, indem mit uninterpretierten Sensordaten gearbeitet wird. Stattdessen sollen die entscheidenden Merkmale und Muster unüberwacht gelernt und wiedererkannt werden. Eine besondere Bedeutung soll dabei zeitlicher Information zugesprochen werden. Zeitliche Muster oder Rhythmen enthalten eine große Menge an Information, die genutzt werden sollte. Gerade bei der Lokalisation kann der zeitliche Ablauf eine große Rolle spielen.

Es ist einleuchtend, dass der Kontext einer Messung oft aufschlussreich ist, wenn es darum geht, Mehrdeutigkeiten auszuschließen. Beispielsweise passiert es oft, dass zwei verschiedene Orte beinahe gleich aussehen. Die Unterscheidung kann dann eventuell nur anhand unter Berücksichtung der näheren Umgebung getroffen werden, insbesondere dem Weg dorthin. Bei landgebundenen Robotern ist der Begriff des zurückgelegten Wegs relativ klar. Anders jedoch bei Untersee-Robotern, Flugrobotern, oder anderen schwebenden, driftenden Maschinen. Hier ist es oft nicht eindeutig festzustellen, ob eine Bewegung im Raum stattgefunden hat, und wenn ja, in welche Richtung mit welcher Geschwindigkeit. Äußere Einflüsse wie Strömungen und Turbulenzen können nicht vorhergesagt werden. Hier ist der zeitliche Verlauf der Sensordaten der einzige Anhaltspunkt.

Das Ziel ist also, zeitliche Muster aus beliebigen Sensor-Eingabedatenströmen unüberwacht einzulernen und wiederzuerkennen. Dabei sollen verschiedene Sensoren unterschiedlicher Charakteristika miteinander kombiniert werden können.

7.2 Ereignissequenzen

Um die hochdimensionalen Sensordaten besser handhaben zu können, wurde der Begriff "Ereignis" eingeführt. Ein Ereignis ist das Auftreten eines bestimmten, unterscheidbaren Merkmals in den Sensordaten. Ein Ereignis ist fest verknüpft mit dem Zeitpunkt, an dem es stattfindet.

Um aus den rohen Sensordaten einen Strom von Ereignissen zu gewinnen, wird ein dynamisches Echtzeit-Clustering-System eingesetzt ([7]). Ein Ereignis ist eine Clustertransition, und wird spezifiziert durch die Kennung des neuen Clusters, und dem Zeitpunkt. Dies entspricht einer Abtastung der Sensordaten mit einer variablen Abtastrate. Die Abtastrate richtet sich nach der Änderungsrate der Daten.

7.3 Das dynamische System

Die zugrundeliegende Idee ist, zeitliche Abfolgen von Ereignissen durch einen gerichteten Graph zu repräsentieren. Die Knoten stehen für Ereignisse, die Kanten tragen die zeitliche Information, d.h. die Zeitspanne, die zwischen den Ereignissen liegt. Im folgenden werden die Knoten als Rezeptoren bezeichnet.

Die Wiedererkennung von Sequenzen basiert auf Aktivitätswellen, die durch den Graphen wandern. Die Rezeptoren sind Verstärkereinheiten, die durch das ihnen entsprechende Ereignis aktiviert werden. Sind sie aktiv, verstärken sie das Signal von den Eingangskanten, und leiten es an die Ausgänge weiter. Ansonsten wird das Eingangssignal gedämpft. Weiterhin soll der Ausgang eines Rezeptors nie ganz null sein, sondern einen geringen positiven Restpegel aufweisen.

Die Kanten funktionieren als einstellbare Verzögerungsleitungen. Ein Eingangssignal wird zeitlich verzögert und aufgeweitet ausgegeben. Sowohl Verzögerungszeit δ und Aufweitung σ sind frei einstellbar, und werden durch einen Adaptionsprozess angepasst. Zusätzlich enthalten Kanten ein Gewicht ω, das durch den später beschriebenen Lernprozess angepasst wird, und einen Flexibilitätsparameter λ, der die Adaptionsfähigkeit der Kante beschreibt.

Die Kanten haben die folgende Übertragungsfunktion:

Definition 7.3.1 *Verzögerungskantenfunktion $d_E(t)$*

$$\begin{aligned} d_E : \Re^+ &\rightarrow [0,1] \\ d_E(t) &\mapsto \omega_E \cdot e^{\left(\frac{-(t-\delta_E)}{\sigma_E}\right)^4} \end{aligned} \tag{7.1}$$

7.3.1 Die Erkennung von zeitlichen Sequenzen

Tritt eine Sequenz von Ereignissen auf, die bereits im Graphen repräsentiert ist, entsteht eine Aktivitätswelle. Dazu kommt es folgendermaßen:

Angenommen, die Ausgangspegel aller Rezeptoren nehmen das Minimum des positiven Restpegels an. Nun tritt eine bereits repräsentierte Ereignissequenz auf. Das erste Ereignis aktiviert einen Rezeptor, der daraufhin, nach Verstärkung des minimalen Eingangspegels, einen geringfügig höheren Ausgangspegel aufweist. Dieser Ausgangspuls wandert entlang der Verzögerungskanten zu den Eingängen anderer Rezeptoren. Einer dieser Rezeptoren wird exakt zu dem Zeitpunkt aktiviert, wenn der Puls an seinem Eingang angekommen ist. Somit wird der Puls weiter verstärkt. Der Vorgang wiederholt sich, der Puls wandert weiter und wird bis zur Sättigungsgrenze verstärkt. Ein durch den Graphen wandernder Puls soll als Aktivitätswelle bezeichnet werden.

Da das Signal auf den Verzögerungskanten aufgeweitet wird, muss die zeitliche Abfolge nicht absolut präzise eingehalten werden. Abweichungen resultieren aber in niedrigeren Amplituden der Aktivitätswelle. Offensichtlich können mehrere Aktivitätswellen simultan im Graphen entstehen. Sie repräsentieren mehrere Hypothesen bezüglich der aktuellen Eingabedaten. Da die Aktivität bei besserer Übereinstimmung stärker wird, stellt die stärkste Welle die beste Hypothese bezüglich der Eingabe dar. Weicht die Eingabe von den Repräsentationen im Graphen ab, werden unpassende Aktivitätswellen gedämpft. Existiert keine Welle einer gewissen Mindeststärke α_{high}, wurde die Eingabe nicht erkannt.

7.3.2 Der Lernprozess

Der Lernprozess besteht aus Aquisition und Adaption. Aquisition findet nur statt, wenn die Eingabe nicht erkannt wurde, also wenn keine starken Aktivitätswellen existieren. Der Adaptionsprozess hingegen kommt bei erkannter Eingabe zum Zuge. Dabei werden die erkannten internen Repräsentationen den aktuellen Eingabedaten angepasst.

Aquisition

Sobald die stärkste Aktivitätswelle im System unter den Schwellwert α_{high} absinkt, beginnt sofort der Aquisitionsprozess. Für jedes auftretende Ereignis wird ein passender Rezeptor erzeugt, und mit dem zuletzt aktiven oder erzeugten Rezeptor verbunden. Dabei wird die neu erzeugte Kante entsprechend der beobachteten Zeitspanne eingestellt. Inzwischen können sich Wellen im System ausbilden. Sobald eine Welle ausreichende Stärke aufweist, wird davon ausgegangen, dass die Eingabe erkannt wurde. Der zuletzt erzeugte Rezeptor wird nun mit dem aktiven Rezeptor der stärksten Welle verknüpft, und der Aquisitionsprozess endet.

Adaption

Bei hoher Aktivität werden die betroffenen Kanten adaptiert, um die interne Repräsentation an die Eingabe anzupassen. Folgende Regeln werden angewandt:

Wenn ein Rezeptor R_j aktiviert wird und ein ausreichendes Aktivitätslevel erreicht, wird die Eingangskante $E_{R_i,R_j} =: E_{ij}$ mit höchster Aktivität ausgewält. Diese Kante wird nach folgenden Regeln adaptiert ($\Delta t = t - t_{R_i}$, t_{R_i} ist der Zeitpunkt, zu dem R_i zuletzt aktiviert wurde):

$$\delta'_E = \delta_E + \lambda_E\left(\tau_E \Delta t - \delta_E\right) \tag{7.2}$$

$$\sigma'_E = \sigma_E + \lambda_E\left(|\tau_E \Delta t - \delta_E| + t_{pmax} - \sigma_E\right) \tag{7.3}$$

$$\omega'_E = \omega_E + \lambda_E\left(1 - \omega_E\right) \tag{7.4}$$

$$\lambda'_E = c_\lambda \lambda_E \tag{7.5}$$

Das Gewicht ω aller Ausgangskanten wird prophylaktisch reduziert:

$$\omega'_E = \omega_E - \lambda_E c_d \omega_E \tag{7.6}$$

Nur die Kanten, die danach erfolgreich einen Rezeptor aktivieren können, werden wieder verstärkt.

7.4 Sensorfusion

Um eine Fusion der Daten von verschiedenen Sensoren zu erreichen, wird für jeden Sensor eine lokaler Graph wie beschrieben aufgebaut und berechnet. Die lokalen Graphen werden mit Querverbindungen verknüpft. Eine Querverbindung entspricht weitgehend den internen Verzögerungskanten.

Wenn in zwei Graphen eine hohe Aktivität erreicht wird, werden Querverbindungen zwischen den Rezeptoren mit der stärksten Aktivität eingefügt. Querverbindungen werden immer von zuerst aktivierten zu einem später aktivierten Rezeptor gezogen, aber nur innerhalb eines Fensters einer begrenzten Anzahl von eintreffenden Ereignissen.

Die Adaption von Querverbindungen unterscheidet sich etwas von der Adaption der internen Kanten. Alle Eingangskanten werden gewichteterweise adaptiert, und nicht nur die aktivste. Damit wird dem Unterschied Rechnung getragen, dass Korrelationen zwischen verschiedenen Sensoren in Ihrer Wichtigkeit gewichtet insgesamt berücksichtigt werden sollen. Innerhalb der Modalitäten interessiert aus Gründen der Abstraktion die beste Repräsentation.

$$\delta'_E = \delta_E + \beta_E(t) \cdot \lambda_E\left(\tau_E \Delta t - \delta_E\right) \tag{7.7}$$

$$\begin{aligned}
\sigma'_E &= \sigma_E + \beta_E(t) \cdot \lambda_E \left(|\tau_E \Delta t - \delta_E| + t_{pmax} - \sigma_E \right) && (7.8)\\
\omega'_E &= \omega_E + \beta_E(t) \cdot \lambda_E \left(1 - \omega_E\right) && (7.9)\\
\lambda'_E &= \beta_E(t) \cdot c_\lambda \lambda_E && (7.10)
\end{aligned}$$

mit

$$\beta_E(t) = e^{\left(\frac{-\left(\left(t-t_{R_i}\right)-\delta_E\right)}{\sigma_E}\right)^4} \alpha_{R_i}(t_{R_i}) \qquad (7.11)$$

7.5 Berechnung der Rezeptor-Aktivität

Definition 7.5.1 *Rezeptoraktivität Sei $E_{S_\psi} := \{E_{ij} = E_{(R_i,R_j)} : R_i \in R_{S_\psi}\}$ die Menge aller Eingangskanten des Rezeptors R_j von der Sensormodalität $S_\psi \in S$, und sei $S_I \in S$ die Menge der involvierten Modalitäten:*

$S_I := \left\{ S_\psi \in S : E_{S_\psi}(R_i, R_j) \neq \emptyset \right\}$.

Die Rezeptoraktivität $\alpha_{R_j}(t)$ berechnet sich nach

$$\begin{aligned}
\alpha_{R_j}(t) &= \phi\left(amp\left(s(t)\right) \left(\varepsilon + \max_{E_i \in E} \{\alpha_{E_i}(t)\} + c_I \cdot \tanh\left(\frac{\gamma}{c_M}\right)\right)\right)\\
\gamma &= \sum_{S_\psi \in S_I} \sum_{E_{ij} \in E_{S_\psi}} \alpha_{E_{ij}}(t) && (7.12)
\end{aligned}$$

ϕ ist eine sättigende Funktion, z.B. $\phi(x) = 1 - e^{-x}$.

Die Funktion

$$f_R(t) = 1 - e^{c_R(E(t) - E(t_R))} \qquad (7.13)$$

beschreibt die zur Verfügung stehende Ausgangsenergie. $E(t)$ ist die Anzahl der zum Zeitpunkt t eingetroffenen Ereignisse, t_R ist der Zeitpunkt, zu dem der Rezeptor R zuletzt aktiviert wurde.

Die Konstanten c_I und c_M kontrollieren die Stärke des Einflusses der Querverbindungen. Der Paramenter c_I beschreibt die Obergrenze des Einflusses, und c_M beschreibt die Mindestanzahl von maximal ausgesteuerten Modalitäten, für die der Einfluss gesättigt ist.

Die Verstärkungsfunktion $amp(s_R(t)) \in [1/c_{damp}, c_a]$ hängt von den Eingangsdaten ab. Sie ist verstärkend (c_a) zu dem Zeitpunkt, an dem ein Ereignis passend zum jeweiligen Rezeptor eintrifft. Ansonsten verhält sich die Funktion dämpfend (c_{damp}).

Definition 7.5.2 *Kantenaktivität: Sei E_{ij} eine Verzögerungskante von Rezeptor R_i nach R_j. Die Kantenaktivität $\alpha_{E_{ij}}(t)$ berechnet sich wie folgt:*

$$\alpha_{E_{ij}}(t) = d_{E_{ij}}\left(t - t_{R_i}\right) \alpha_{R_i}(t_{R_i}) \qquad (7.14)$$

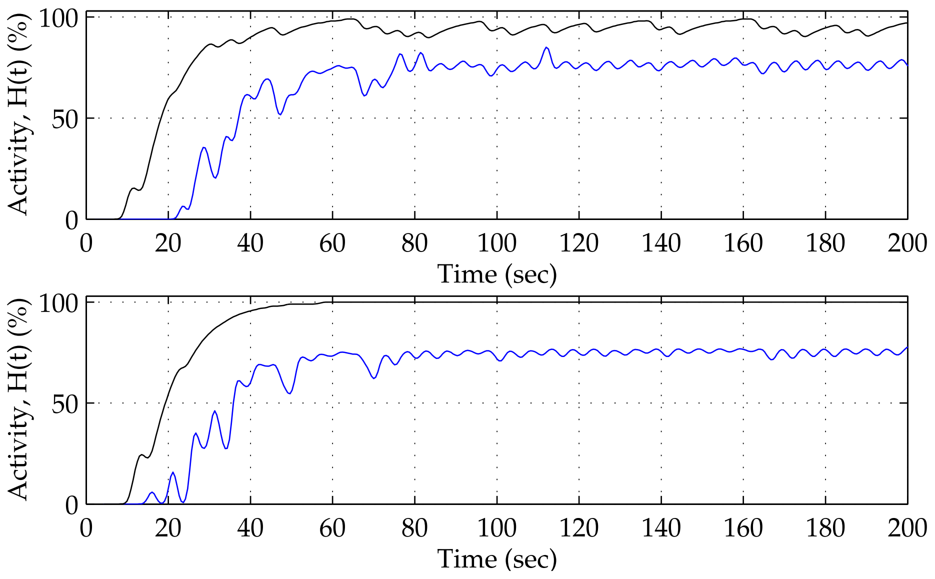

Figure 7.1: Aktivitätslevel (blau, obere Kurve) und Vorhersagerichtigkeit $P(t)$ (schwarz) für das zeitlich frühere System (oben) und das zeitverzögerte System (unten).

(t_{R_i} ist der Zeitpunkt, zu dem R_i zuletzt aktiviert wurde)

7.6 Experimente

Es wurden diverse Experimente mit simulierten und realen Daten durchgeführt. Eine ausführliche Beschreibung befindet sich in der englischen Version. Hier sollen nur beispielhaft zwei Experimente herausgegriffen werden.

7.6.1 Sensorfusion in der Simulation

Dieses Experiment soll die prinzipielle Funktionalität des Systems untersuchen. Es wurden zwei simulierte Ereignisströme erzeugt. Es handelt sich um eine identische Sequenz, die zeitversetzt in die beiden dynamischen Systeme eingegeben wird. Die Sequenz besteht aus sich wiederholenden Teilsequenzen, wobei eine zufällige Bifurkation auftritt. Als Messwert dient die Fähigkeit der Systeme, vorherzusagen, welcher Rezeptor als nächstes die höchste Aktivität erreichen wird. Das beinhaltet eine Vorhersage des nächsten Ereignisses samt Zeitpunkt unter Berücksichtigung des Kontextes.

Das System, das die Sequenz früher beobachtet, kann die Bifurkation nicht vorhersagen. Somit wird der Prozentsatz richtiger Vorhersagen unter dem Maximum von 100% liegen. Das zweite System, welches die Sequenz zeitverzögert erhält, kann von den Informationen profitieren, die es über die erzeugten Querverbindungen erhält. Es ist somit in der Lage, absolut korrekte Vorhersagen zu treffen. Die Ergebnisse sind in Abbildung 7.1 zu sehen.

Das Experiment zeigt sowohl, dass das Verfahren lokal funktioniert, als auch den positiven Einfluss der Sensorfusion.

7.6.2 Experiment mit echten Sensordaten

Zu diesem Experiment wurde ein Nomad XR4000 Roboter eingesetzt. Es handelt sich um eine holonomische Plattform, die mit einem SICK Laser-Entfernungssensor, Sonar und diversen anderen Sensoren ausgestattet ist. Der Roboter fuhr autonom einen L-förmigen Teil einer Büroumgebung wiederholt ab. Hierbei kam ein einfacher Explorationsalgorithmus zum Einsatz, der den eingebauten Lasersensor nutzte. Die gewählten Pfade waren in den Wiederholungen ähnlich, jedoch nie absolut gleich.

Für das Experiment wurden zwei dynamische Systeme eingesetzt, die ihren Eingabestrom aus zwei identischen Lasersensoren erhielten. Aus den Laser-Entfernungsdaten wurde mit der beschriebenen Cluster-Methode eine Ereignis-Sequenz erzeugt.

Einer der verwendeten Lasersensoren war der im Roboter eingebaute. Der andere Sensor war stationär auf am längeren Ende der L-förmigen Sektion aufgestellt. Dieser Sensor konnte den Roboter zeitweise beobachten, wenn er sich nicht ausserhalb des Sichtfeldes befand. Es kann somit nur zeitweise ein Effekt der Sensorfusion entwickeln. Das Experiment wurde mit den identischen Daten aus einem Durchlauf einmal mit und einmal ohne Verwendung von Sensorfusion durchgeführt.

Wie deutlich in Abbildung 7.2 zu sehen ist, liefert der externe Sensor nur intervallweise Ereignisse. Erkennbar ist das an den regelmäßigen Unterbrechungen, die auftraten wenn der Roboter außer Sichtweite war. Sehr auffällig ist der deutliche Anstieg der Erkennungs- und Vorhersagequalität des internen Sensors, wenn Daten vom externen Sensor vorliegen. Dieses Experiment belegt deutlich den positiven Einfluss der Sensorfusion. Außerdem zeigt es, wie problemlos Sensordatenströme unterschiedlicher Charakteristika und zeitlicher Verhalten kombiniert werden können. Korrelationen zwischen den unterschiedlichen Sensoren werden identifiziert und genutzt, wo sie vorhanden sind.

7.7 Zusammenfassung der Ergebnisse

Es wurde ein Verfahren vorgestellt, das Ereignissequenzen unter Berücksichtigung der zeitlichen Eigenschaften lernen und wiedererkennen kann. Es verar-

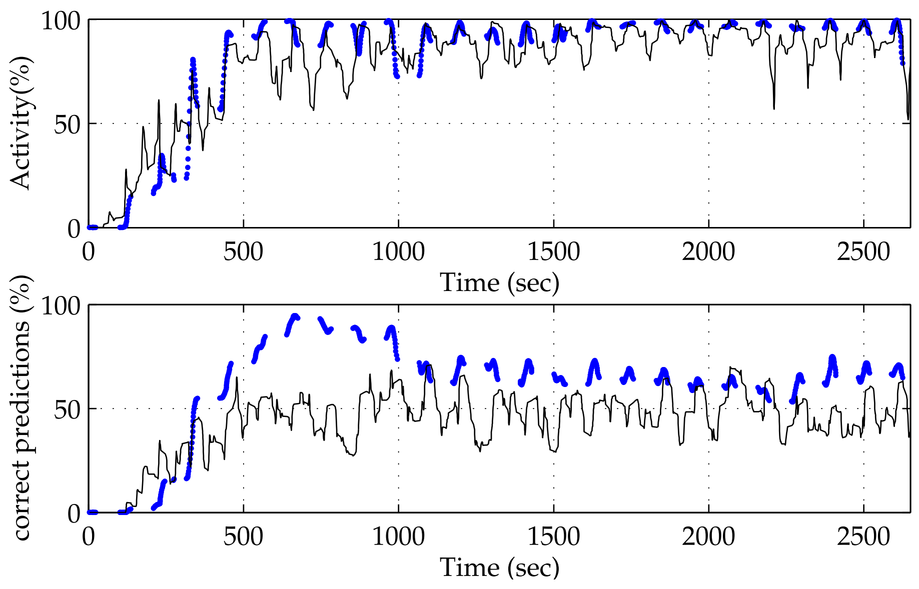

Figure 7.2: Experiment mit einem internen und einem stationären, externen Lasersensor. Die dünne schwarze Kurve zeigt die Resultate des internen Sensors, die dicke blaue Kurve beschreibt den externen Sensor.

beitet dabei Eingabedaten aus verschiedenen Quellen mit beliebigen Abtastfrequenzen und Charakteristika.

Das Verfahren wurde als dynamisches, verteiltes Echtzeitsystem implementiert. Die Funktionalität wurde in zahlreichen Experimenten mit simulierten und realen Sensordaten belegt. Es wurde gezeigt, dass das vorgestellte Verfahren zur topologischen simultanen Kartographierung und Positionsbestimmung auf mobilen Roboterplattformen eingesetzt werden kann. Die geringe Berechnungskomplexität ermöglicht Echtzeitbetrieb, und die Auslegung als verteiltes System bietet vielfältige Einsatzmöglichkeiten und große Flexibilität. Das System passt sich dynamisch an die spatio-temporale Komplexität der Eingabedaten an, und es gibt keine festen Limitierungen der Anzahl an unterschiedlichen Ereignissen oder der Größe der internen Repräsentation (abgesehen vom Arbeitsspeicher des verwendeten Computers). Diese Größen müssen nicht vorab festgelegt werden, sondern wachsen dynamisch entsprechend der Eingabe.

Das System lernt neue Sequenzen unmittelbar und sehr schnell. Bei perfekter Wiederholung wird eine Sequenz bereits bei der ersten Wiederholung erkannt. Bei Versuchen mit realen Sensordaten reichten 3-4 Wiederholungen, damit auftretende Variationen eingelernt, und die Sequenz mit großer Sicherheit erkannt werden konnte.

Eingesetzt zur simultanen Kartographierung und Positionsbestimmung, eröffnet es neue Möglichkeiten auf unkonventionellen Plattformen wie Untersee-Roboter.

Gerade unter Wasser sind die Messergebnisse von Sonar- und Lasersensoren schwer modellierbar und sehr abhängig von den Umgebungsbedingungen. In diesen schwierig modellierbaren Umgebungen passt sich das System selbstständig an die Gegebenheiten an. Es ist sogar möglich, beliebige unkonventionelle Sensoren einzusetzen und zu kombinieren, z.B. Sensoren für Wasserdruck, elektrische Leitfähigkeit, das einfallende Lichtspektrum oder die optische Qualität des Wassers. Diese Sensoren können einen Beitrag zur Positionsbestimmung leisten. In vielen anderen Verfahren wäre eine Nutzung dieser Sensoren nur schwer möglich. Es sind zahlreiche Einsatzmöglichkeiten in anderen Bereichen ebenso denkbar. Hier eröffnen sich zahlreiche Möglichkeiten für weitere Untersuchungen.

Chapter 8

Appendix B: Selection of source codes

8.1 Specification of the generic protected list

```
-- This package implements a double-linked generic list --
-- List entries containing List Elements are created as needed --
-- and kept for recycling at removal --
--
-- This implementation provides transparent locking on writes by
-- counting references on the list entries.

-- If elements are removed, they remain in the list, until they are
-- not referenced any more. Then they are removed finally. Until then,
-- existing references are still valid, but no new references to this
-- element can be created.

-- This allows concurrent access to the same list by several tasks.

-- IMPORTANT: For proper functionality, the procedure
--
-- Release(ActualList:ListDescriptor)
--
-- MUST be used before disposing a ListDescriptor!
-- otherwise, the element specified by the enumerator of the
-- listdescriptor  remains locked for ever.

with Ada.Unchecked_Deallocation;

generic
   type ListType is private;

package GenericList is

   type ListEntryType is private;
   type ListEntryAccess is access ListEntryType;

   type ListDescriptor is private;

   type ListHandle is private;



   protected Sec is

      -- creates a new, empty list
      function CreateList return ListDescriptor;

      -- deletes a list and recycles the rest
      procedure DisposeList(OldDescriptor:in out ListDescriptor);

      -- creates a handle for the given list
      -- the Enumerator is invalid
      function CreateHandle(ActualList:in ListDescriptor) return ListHandle;
      -- equivalent
      function CreateHandle(ActualList:in ListHandle) return ListHandle;

      -- duplicates a list handle; the enumerator has the same value
      function DuplicateHandle(ActualList:ListHandle) return ListHandle;

      -- Adds an element to the list; a new ListEntryType is allocated
      -- the handle refers to the newly inserted element afterwards
      procedure Add(Handle:in out ListHandle; Element: in ListType);
      -- Inserts an element at the position specified by the handle.
      -- the new Element will be inserted before the current element.
      -- if the handle is invalid, the element will be appended
      -- the handle refers to the newly inserted element afterwards
      procedure Insert(Handle:in out ListHandle; Element: in ListType);

      -- removes current active entry (Entry given by handle)
      -- The Enumerator is set to null
      procedure Remove(Handle: in out ListHandle);

      -- marks the current entry for deletion.
      -- The entry is deleted, when the last reference leaves it,
      -- and cannot be entered any more.
      procedure MarkForDeletion(Handle:in out ListHandle);

      -- Removes all Elements from the list
      -- uses Remove
      procedure RemoveAll(Handle:in out ListHandle);

      -- This procedure overwrites the specified element
      -- it is non-blocking; the change is visible to all others referring to
      -- this element
      procedure OverwriteElement(Handle:in ListHandle; Element:in ListType);

      -- moves the handle to the specified element, if existent;
      -- otherwise the Handle will be invalid
      procedure GoToElement(Handle:in out ListHandle; Element:in ListType);

      -- returns true if the Element specified by enumerator equals given Element
      function EntryEquals(Handle:ListHandle; Element:ListType) return Boolean;

      -- moves the handle to the first valid entry of the list
      procedure JumpToHead(Handle:in out ListHandle);

      -- moves the handle to the last valid entry of the list
      procedure JumpToTail(Handle:in out ListHandle);

      -- Releases the current Element by setting the enumerator to null
      -- should be used after every transaction to free the List for other tasks
      -- must be used before disposing a ListHandle!
      procedure Release(Handle:in out ListHandle);

      -- Delivers the element specified by the handle
      -- If the handle is invalid, a Constraint_Error will be raised.
      -- to avoid this, use IsValid or EndExceeded before.
      function GetCurrent(Handle:ListHandle) return ListType;

      -- Moves forward in the list.
      -- If handle is invalid, JumpToHead is called
      procedure StepForward(Handle:in out ListHandle);

      -- Moves backwards in the list.
      -- If handle is invalid, JumpToTail is called
      procedure StepBack(Handle:in out ListHandle);

      -- Returns true if list contains no entries
      function IsEmpty(Handle:ListHandle) return Boolean;

      -- Returns true if handle is pointing to the Head of the list
      function IsFirstEntry(Handle:ListHandle) return Boolean;

      -- Returns true if handle is pointing to the Tail of the list
      function IsLastEntry(Handle:ListHandle) return Boolean;

      -- Returns true, if handle is null - negation of IsValid
      function EndExceeded(Handle:ListHandle) return Boolean;

      -- Returns true, if handle is valid
      function IsValid(Handle:ListHandle) return Boolean;


   private

      entry Wait;


   end Sec;
   function Size(Handle:ListHandle) return Long_Integer;

private

   Recycle:ListEntryAccess; -- List of created, but unused ListEntry-Objects
   SomethingChanged:Boolean;

   type ListDescriptor is
      record
         Guardian:ListEntryAccess;
```

```
     end record;


   type ListHandle is
      record
         Guardian:ListEntryAccess;
         Enumerator:ListEntryAccess;            -- points to current active entry
      end record;



   type ListEntryType is
      record
         Element:ListType;
         Prev, Next:ListEntryAccess;
         references:Integer range 0..Integer'Last;      -- counts the number of references to this element
         Deleted:Boolean:=False;
      end record;

   procedure DisposeEntry is
      new Ada.Unchecked_Deallocation(ListEntryType, ListEntryAccess);


end GenericList;
```

8.2 Specification of the communication module

8.2.1 The communication package

```
with TCPIdentificationIntf; use TCPIdentificationIntf;
with TCPMonitor;

package TCPCommIntf is

   subtype IDType is AbstractID;

   InternalID, PartitionServer:IDType;
   MyID: NumID:=0;

   package PrivateMonitor is new TCPMonitor;


   procedure SendEvent(Receiver:AbstractID; Content:in Message; Sender:in NumID:=MyID);

   procedure SendEvent(Receiver: in NumID; Content:in Message; Sender:in NumID:=MyID);

   -- Broadcast to all registered members
   procedure LocalBroadCast(Content:Message; Sender:in NumID:=MyID);

   procedure SetInternalAdress(LocalName:String; ListenPort:Integer);

   procedure Register(PServerName:String; PServerPort:Integer);


   procedure ShutDownCommunication;

   ProgramActive:Boolean:=True;

end TCPCommIntf;
```

8.2.2 The message retrieval monitor

```
with TCPIdentificationIntf; use TCPIdentificationIntf;
generic
package TCPMonitor is

   protected Monitor is
      entry Listen(Sender: out NumID; Content:out Message);
      entry WaitingQueue(Sender:out NumID; Content:out Message);
      procedure SetEvent(Sender:in NumID; Content:in Message);
      procedure Shutdown;
   end Monitor;
end TCPMonitor;
```

8.2.3 The identification specification

```
with MessageDef; use MessageDef;
with Ada.Real_Time; use Ada.Real_Time;

package TCPIdentificationIntf is

   type MessageTags is (NormalMsg, NewMemberMsg, AckMsg, BroadcastMsg, IDMsg);

   subtype NumId is Long_Integer;


   subtype AdressString is String(1..40);
   subtype TextString is String(1..10);

   type AbstractID is record
      ID:NumID:=-1;
      Adress:AdressString:="                                        ";
      Port:Integer:=-1;
   end record;

   type CompleteID is record
      IDObj:AbstractID;
      ID:NumID;
   end record;


   type Message is record
      Content:GenMessage;
      Text:TextString:="          ";
      MsgType:MessageTags:=NormalMsg;
      Sender:AbstractID;
      TimeStamp:Duration:=0.0;
   end record;


   function StringImage(Input:in AbstractID) return String;
   function StringImage(Input:in Message) return String;
   function ToAbstractID(Input:in String)return AbstractID;
   function ToMessage(Input:String) return Message;

   AbstractIDWidth, MessageWidth:Integer;


end TCPIdentificationIntf;
```

8.3 Mapping constants

```
with GraphData; use GraphData;

package Constants is

   INIT_TEMP_PRECISION: RealNumber:=   0.5 ; -- initial Factor Sigma/DeltaT
                                            -- low factor: high temporal precision
   INIT_CROSSEDGE_TEMP_PRECISION: RealNumber:=   1.0 ; -- initial Factor Sigma/DeltaT for Cross Edges

   MAX_TEMP_PRECISION:  NormRange :=  0.1 ; -- absolute lower bound for sigma (in sec)
   INITIAL_LAMBDA:      NormRange :=  0.5 ;
   INITIAL_EDGE_WEIGHT: NormRange :=  0.7 ;
   INITIAL_CROSSEDGE_WEIGHT:NormRange:=0.1;

   C_LAMBDA:            NormRange :=  0.1;  -- Learn rate reduction factor
   C_D:                 NormRange :=  0.03; -- inactive edge decay factor (decay in percent)
   C_T:                 NormRange :=  0.4 ; -- time warp adaptation factor;
   EPSILON:             NormRange :=  0.02; -- self activation term
   C_A:                 RealNumber:=  4.5 ; -- Amplification factor - must be >1
                                            -- (1 means original strength - no amplification)
   C_DAMP:              RealNumber:=  2.0 ; -- Damping factor for imaginated events - must be >1
                                            -- (1 is no damping - do not use 1!)

   CROSSEDGE_C_D:       NormRange :=  0.1;  -- inactive cross edge decay factor (decay in percent)
   CROSSEDGE_BUNDLESIZE:Integer   :=  3;    -- number of steps back in time at crossedge creation
   CROSSEDGE_INFLUENCE: RealNumber:=  1.3 ; -- maximal additive influence from cross edges
                                            -- should be <C_DAMP
                                            -- (internal edges are always 100%)
   MODALITIES_TO_SATURATE:RealNumber:=2.0 ; -- Number of modalities necessary to saturate
                                            -- cross edge influence
   CROSSEDGE_ADAPTATION:Boolean   :=  True; -- Switch for adaption of CrossEdges

   RECHARGING_RATE:     NormRange :=  0.2 ; -- Speed of receptor recharging after firing
   LOW_ACT:             NormRange :=  0.15; -- low activity threshold
   HIGH_ACT:            NormRange :=  0.5 ; -- recognition activity threshold
   PM_HALFTIME:         Long_Float:=  4.0 ; -- Prediction measure smoothing parameter
   MaxConcHypotheses:   constant Integer:=5;

   PREDICTION_DEBUG_INFO:Boolean  :=  False;

end Constants;
```

8.4 Main module of the mapping system (graphmanager)

8.4.1 Specification file “graphmanager.ads”

```
with GenericList;
with GraphData; use GraphData;
with Ada.Real_Time; use Ada.Real_Time;

with TCPIdentificationIntf;  use TCPIdentificationIntf;
with TCPCommIntf; use TCPCommIntf;
with Constants; use Constants;

generic
package GraphManager is

   type Edge;
   type EdgeAccess is access Edge;

   type Receptor;
   type ReceptorAccess is access Receptor;

   type ActivityListType is array (1..MaxConcHypotheses) of ReceptorAccess;


   package EdgeList is
      new GenericList(ListType=>EdgeAccess);

   type CrossEdgesDescriptor is record
      ModID:Long_Integer;
      Edges:EdgeList.ListDescriptor;
   end record;

   package CrossEdgeList is
      new GenericList(ListType=>CrossEdgesDescriptor);


   package ReceptorList is
      new GenericList(ListType=>ReceptorAccess);

   type Event is record
      TimeStamp:GlobalTime;
      EventClass:Long_Integer;
      Similarity:NormRange;
      PosX, PosY, Theta:Float;
   end record;


   type Edge is
      record
         Source, Target: ReceptorAccess:=null;
         Weight: NormRange:=1.0;
         Lambda: NormRange:=0.5;
         Tau:RealNumber:=1.0;
         DeltaT, Sigma:TimeInterval;
         CrossEdge:Boolean:=False;
      end record;


   type Receptor is
      record
         InputEdges:EdgeList.ListDescriptor;            -- incoming internal edges
         InCrossEdges:CrossEdgeList.ListDescriptor;     -- incoming crossedges
         OutputEdges: EdgeList.ListDescriptor;          -- outgoing internal edges
         Activity, Stimulation:NormRange:=0.0;
         InvSensors:Integer:=1;
         TriggerEvent:Event;                            -- Event which last triggered this receptor
         ID, SeqID:Long_Integer:=0;
         EventCount:Long_Float:=0.0;
         CrossMirror:Boolean:=False;                    -- true, if this receptor is just a mirror
```

```
                                                   -- of a receptor from another modality
         ModID:Long_Integer:=-1;
      end record;


   type EventDescriptor is record
      EventClass:Long_Integer;
      CorrReceptors:ReceptorList.ListDescriptor;
   end record;


   package EventList is
      new GenericList(ListType=>EventDescriptor);


   type ModalityDescriptor is record
      ModID:Long_Integer;
      Receptors:ReceptorList.ListDescriptor;
      ClockSkew:Duration;
      LastMostActive:ReceptorAccess;
   end record;

   package ModalityList is
      new GenericList(ListType=>ModalityDescriptor);

   ProgramActive:Boolean:=True;
   HypothesisActivity:NormRange:=0.0;

   -- the local input interface
   protected InputHandler is
      -- this is the entry for local sensor events
      entry Notify(LastEvent:in Event);
      entry GetEvent(LastEvent:out Event);
   end InputHandler;

   procedure Save;

   -- (internal) processes local sensor input
   task InputCalculations;

   task DreamTask is
      entry Suspend;
      entry Notify;
   end DreamTask;



   task ConcurrentActivityHandler is
   end ConcurrentActivityHandler;


end GraphManager;
```

8.4.2 Implementation file “graphmanager.adb”

```
with Ada.Numerics.Long_Elementary_Functions; use Ada.Numerics.Long_Elementary_Functions;
with GenericList;

with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use  Ada.Integer_Text_IO;
with Ada.Float_Text_IO; use  Ada.Float_Text_IO;
with Ada.Long_Integer_Text_IO; use  Ada.Long_Integer_Text_IO;
with Ada.Exceptions;                    use Ada.Exceptions;
with Ada.Task_Identification;           use Ada.Task_Identification;
with TagSupport;   use TagSupport;
with MessageDef; use MessageDef;

with Ada.Command_Line; use Ada.Command_Line;

package body GraphManager is

   ActivityList, OldActivityList:ActivityListType;

   MyEventList:EventList.ListDescriptor;

   ConcurrentActivityList:ModalityList.ListDescriptor;
   ConcurrentActivityTaskReady:Boolean:=False;

   ReceptorGarbage:ReceptorList.ListDescriptor;
   EdgeGarbage:EdgeList.ListDescriptor;

   CurrentReceptorID, CurrentSeqID, EventCounter:Long_Integer:=0;

   InternalLastEvent:Event;
   EventArrived:Boolean;
```

```
function GetGlobalTime return Time is
begin
   return Clock;
end;



-- Monitor for input stream handling
protected body InputHandler is

   -- non-blocking input entry
   entry Notify(LastEvent:in Event) when True is
   begin
      if InternalLastEvent.EventClass/=LastEvent.EventClass then
         InternalLastEvent:=LastEvent;
         EventCounter:=EventCounter+1;
         EventArrived:=True;
      end if;
   end Notify;

   -- blocking event retrieval
   entry GetEvent(LastEvent:out Event)
   when (EventArrived and ConcurrentActivityTaskReady) or not ProgramActive is
   begin
      LastEvent:=InternalLastEvent;
      EventArrived:=False;
   end GetEvent;

end InputHandler;


function GetRecycledReceptor return ReceptorAccess is
   Handle :ReceptorList.ListHandle;
   EdgeHandle:EdgeList.ListHandle;
   CrossHandle:CrossEdgeList.ListHandle;
   NewRec:ReceptorAccess;
begin
   Handle:=ReceptorList.Sec.CreateHandle(ReceptorGarbage);
   ReceptorList.Sec.JumpToHead(Handle);
   if ReceptorList.Sec.IsValid(Handle) then
      NewRec:=ReceptorList.Sec.GetCurrent(Handle);
      EdgeHandle:=EdgeList.Sec.CreateHandle(NewRec.InputEdges);
      EdgeList.Sec.RemoveAll(EdgeHandle);
      EdgeHandle:=EdgeList.Sec.CreateHandle(NewRec.OutputEdges);
      EdgeList.Sec.RemoveAll(EdgeHandle);

      CrossHandle:=CrossEdgeList.Sec.CreateHandle(NewRec.InCrossEdges);
      CrossEdgeList.Sec.RemoveAll(CrossHandle);
      ReceptorList.Sec.Remove(Handle);
   else
      NewRec:=new Receptor;
      NewRec.InputEdges:=EdgeList.Sec.CreateList;
      NewRec.OutputEdges:=EdgeList.Sec.CreateList;
      NewRec.InCrossEdges:=CrossEdgeList.Sec.CreateList;
   end if;
   -- always release Handles!!
   ReceptorList.Sec.Release(Handle);
   NewRec.InvSensors:=1;

   return NewRec;
end GetRecycledReceptor;


function CreateReceptor(RecType:EventDescriptor) return ReceptorAccess is
   Handle :ReceptorList.ListHandle;
   NewRec:ReceptorAccess;
begin
   NewRec:=GetRecycledReceptor;
   CurrentReceptorID:=CurrentReceptorID+1;
   NewRec.ID:=CurrentReceptorID;
   NewRec.TriggerEvent.EventClass:=RecType.EventClass;
   NewRec.EventCount:=Long_Float(EventCounter);
   -- register new receptor in corresponding event receptor table
   Handle:=ReceptorList.Sec.CreateHandle(RecType.CorrReceptors);
   ReceptorList.Sec.Add(Handle, NewRec);
   ReceptorList.Sec.Release(Handle);
   return NewRec;
end CreateReceptor;


function CreateEdge(From: ReceptorAccess; To:ReceptorAccess) return EdgeAccess is
   Handle:EdgeList.ListHandle;
   CrossHandle:CrossEdgeList.ListHandle;
   ModDescr:CrossEdgesDescriptor;
   NewEdge:EdgeAccess;
   NewDeltaT, NewSigma:TimeInterval;
begin
```

```
      if From/=To and not To.CrossMirror then
         NewDeltaT:=TimeInterval(To_Duration(To.TriggerEvent.TimeStamp-From.TriggerEvent.TimeStamp));
         NewSigma:=INIT_TEMP_PRECISION*NewDeltaT;

         -- check if edge already exists
         Handle:=EdgeList.Sec.CreateHandle(From.OutputEdges);
         EdgeList.Sec.JumpToHead(Handle);
         while EdgeList.Sec.IsValid(Handle) loop
            NewEdge:=EdgeList.Sec.GetCurrent(Handle);
            if NewEdge.Target.ID=To.ID then
               if NewEdge.DeltaT>NewDeltaT-NewSigma and
                 NewEdge.DeltaT<NewDeltaT+NewSigma then
                  -- do not create edge!
                  EdgeList.Sec.Release(Handle);
                  return null;
               end if;
            end if;
            EdgeList.Sec.StepForward(Handle);
         end loop;
         EdgeList.Sec.Release(Handle);

         Handle:=EdgeList.Sec.CreateHandle(EdgeGarbage);
         EdgeList.Sec.JumpToHead(Handle);
         if EdgeList.Sec.IsValid(Handle) then
            NewEdge:=EdgeList.Sec.GetCurrent(Handle);
            EdgeList.Sec.Remove(Handle);
         else
            NewEdge:=new Edge;
         end if;
         EdgeList.Sec.Release(Handle);
         NewEdge.DeltaT:=NewDeltaT;
         NewEdge.Sigma:=NewSigma;
         NewEdge.Source:=From;
         NewEdge.Target:=To;
         NewEdge.Weight:=1.0;
         NewEdge.Lambda:=INITIAL_LAMBDA;
         NewEdge.Tau:=1.0;

         NewEdge.CrossEdge:=From.CrossMirror;

         -- connect edge to receptors
         Handle:=EdgeList.Sec.CreateHandle(From.OutputEdges);
         EdgeList.Sec.Add(Handle, NewEdge);
         EdgeList.Sec.Release(Handle);

         if NewEdge.CrossEdge then
            -- this is a cross-edge
            -- try to find corresponding modality
            CrossHandle:=CrossEdgeList.Sec.CreateHandle(To.InCrossEdges);
            CrossEdgeList.Sec.JumpToHead(CrossHandle);
            while CrossEdgeList.Sec.IsValid(CrossHandle)
              and then CrossEdgeList.Sec.GetCurrent(CrossHandle).ModID/=From.ModID
            loop
               CrossEdgeList.Sec.StepForward(CrossHandle);
            end loop;
            -- if not found, create new entry
            if not CrossEdgeList.Sec.IsValid(CrossHandle) then
               ModDescr.ModId:=From.ModID;
               ModDescr.Edges:=EdgeList.Sec.CreateList;
               CrossEdgeList.Sec.Add(CrossHandle, ModDescr);
            else
               ModDescr:=CrossEdgeList.Sec.GetCurrent(Crosshandle);
            end if;
            -- and add edge into corresponding modality entry.
            Handle:=EdgeList.Sec.CreateHandle(ModDescr.Edges);
            EdgeList.Sec.Add(Handle, NewEdge);
            EdgeList.Sec.Release(Handle);
            -- update counter for involved sensor modalities
            To.InvSensors:=Integer(CrossEdgeList.Sec.Size(CrossHandle));
            CrossEdgeList.Sec.Release(CrossHandle);
         else
            -- it's an internal edge
            Handle:=EdgeList.Sec.CreateHandle(To.InputEdges);
            EdgeList.Sec.Add(Handle, NewEdge);
            EdgeList.Sec.Release(Handle);
         end if;
--         Put(" T:");
--         Put(Integer(NewEdge.DeltaT*10.0),3);
         return NewEdge;
      end if;
      return null;
   end CreateEdge;


   procedure PrintActivityList is
   begin
      Put(" HC:");
      Put(Integer(HypothesisActivity*100.0), 4);
      Put(" | ");

      for I in 1..MaxConcHypotheses loop
         if ActivityList(i)/=null then
            Put("I");
            Put(Integer(ActivityList(I).ID),3);
            Put(Integer(ActivityList(I).Activity*100.0),4);
            Put("%");
         else
            Put("   -     ");
         end if;
         Put("|");
      end loop;
   end;


   protected Sec is
      procedure Add2ActivityList(ActReceptor:ReceptorAccess);
      procedure RemoveFromList(I:Integer);
      procedure ClearActivityList;
      -- pure receptor activity calculation without adaption
      procedure CalcReceptorActivity(Rec:access Receptor);
      -- calc of Rec.Activity with adaption and learning
      procedure DoReceptorCalculation(Rec:access Receptor);
      procedure DoReceptorAdaptation(Rec:ReceptorAccess);
      procedure SaveGraph(File: in File_Type);
      procedure SaveAsFilterLayout(File:in File_Type);
   end Sec;

   protected body Sec is
      procedure Add2ActivityList(ActReceptor:ReceptorAccess) is
         Index:Integer;
      begin
         -- list contains the most active receptors sorted descending
         Index:=MaxConcHypotheses;
         if ActReceptor.Activity>LOW_ACT then
            while Index>0 and then ((ActivityList(Index)=null
                                     or else ActReceptor.Activity>ActivityList(Index).Activity))
            loop
               if Index<MaxConcHypotheses then
                  -- shift backwards to make room
                  ActivityList(Index+1):=ActivityList(Index);
               end if;
               Index:=Index-1;
            end loop;
            if Index<MaxConcHypotheses then
               -- insert at found position
               ActivityList(Index+1):=ActReceptor;
            end if;
         end if;
      end Add2ActivityList;

      procedure RemoveFromList(I:Integer) is
      begin
         for Index in I..MaxConcHypotheses-1 loop
            ActivityList(Index):=ActivityList(Index+1);
         end loop;
         ActivityList(MaxConcHypotheses):=null;
      end;

      procedure ClearActivityList is
      begin
         OldActivityList:=ActivityList;
         for I in 1..MaxConcHypotheses loop
            ActivityList(I):=null;
         end loop;
      end ClearActivityList;


      function CalcEdgeActivity(E:in Edge) return NormRange is
         D:NormRange;
         T:TimeInterval;
      begin
         T:=E.Tau*TimeInterval(To_Duration(E.Target.TriggerEvent.TimeStamp-
                                           E.Source.TriggerEvent.TimeStamp));
         D:=Exp(-((T-E.DeltaT)/E.Sigma)**4);
         return (E.Weight*E.Source.Activity*D);
      end CalcEdgeActivity;

      procedure CalcReceptorActivity(Rec:access Receptor) is
         EdgeActivity:NormRange;
         EdgeSum, TmpActivity:RealNumber;
         EdgeListHandle:EdgeList.ListHandle;
         ActEdge:EdgeAccess;
      begin
         -- go through all input edges
         EdgeListHandle:=EdgeList.Sec.CreateHandle(Rec.InputEdges);
         EdgeList.Sec.JumpToHead(EdgeListHandle);
         EdgeSum:=0.0;
         while EdgeList.Sec.IsValid(EdgeListHandle) loop
            ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
            EdgeActivity:=CalcEdgeActivity(ActEdge.all);
            EdgeSum:=EdgeSum+EdgeActivity;
            EdgeList.Sec.StepForward(EdgeListHandle);
         end loop;
         TmpActivity:=Rec.Stimulation*(1.0+C_A)*(EPSILON+EdgeSum/RealNumber(Rec.InvSensors));
         Rec.Activity:=1.0-Exp(-TmpActivity);
      end;

      -- here all the calculation, learning and adaptation for one receptor is done.
      procedure DoReceptorCalculation (Rec:access Receptor) is

         MaxActivity, EdgeActivity, FiringEnergy:NormRange;
         EdgeSum, NewTau, TmpActivity:RealNumber;
         EdgeListHandle:EdgeList.ListHandle;
         CrossEdgeListHandle:CrossEdgeList.ListHandle;
         ActEdge,MaxEdge:EdgeAccess;
      begin

         -- go through all cross edge lists
         CrossEdgeListHandle:=CrossEdgeList.Sec.CreateHandle(Rec.InCrossEdges);
         CrossEdgeList.Sec.JumpToHead(CrossEdgeListHandle);
         MaxActivity:=0.0;

         while CrossEdgeList.Sec.IsValid(CrossEdgeListHandle) loop
            -- go through all cross edges
            EdgeListHandle:=EdgeList.Sec.
              CreateHandle(CrossEdgeList.Sec.GetCurrent(CrossEdgeListHandle).Edges);
            EdgeList.Sec.JumpToHead(EdgeListHandle);
            MaxActivity:=0.0;
            EdgeSum:=0.0;
            -- select maximum edge activity
            while EdgeList.Sec.IsValid(EdgeListHandle) loop
               ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
               EdgeActivity:=CalcEdgeActivity(ActEdge.all);
               if EdgeActivity>MaxActivity then
                  MaxActivity:=EdgeActivity;
               end if;
               EdgeList.Sec.StepForward(EdgeListHandle);
            end loop;
            -- and add edge with maximum activity of this modality to EdgeSum
            EdgeSum:=EdgeSum+MaxActivity;

            EdgeList.Sec.Release(EdgeListHandle);
            CrossEdgeList.Sec.StepForward(CrossEdgeListHandle);
         end loop;
         CrossEdgeList.Sec.Release(CrossEdgeListHandle);


         -- go through all input edges
         EdgeListHandle:=EdgeList.Sec.CreateHandle(Rec.InputEdges);
         EdgeList.Sec.JumpToHead(EdgeListHandle);
         MaxActivity:=0.0;

         while EdgeList.Sec.IsValid(EdgeListHandle) loop
            ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
            EdgeActivity:=CalcEdgeActivity(ActEdge.all);
            if EdgeActivity>MaxActivity then
               MaxActivity:=EdgeActivity;
               MaxEdge:=ActEdge;
            end if;
            EdgeList.Sec.StepForward(EdgeListHandle);
         end loop;
         EdgeList.Sec.Release(EdgeListHandle);
         -- and add maximum activity of internal edges
         EdgeSum:=EdgeSum+MaxActivity;


         FiringEnergy:=1.0-Exp(Long_Float(RECHARGING_RATE)*(Rec.EventCount-Long_Float(EventCounter)));
         TmpActivity:=Rec.Stimulation*FiringEnergy*
           (1.0+C_A)*(EPSILON+EdgeSum/RealNumber(Rec.InvSensors));
         Rec.Activity:=1.0-Exp(-TmpActivity);

         if Rec.Activity>HIGH_ACT then
            Rec.EventCount:=Long_Float(EventCounter);
         end if;
         -- do time-warp adaptation for input edge with maximum activity
         if MaxEdge/=null and Rec.Activity>LOW_ACT then
            -- adapt time warping factor
            NewTau:=MaxEdge.Tau+C_T*
              (MaxEdge.DeltaT/
               TimeInterval(To_Duration(MaxEdge.Target.TriggerEvent.TimeStamp-
                                        MaxEdge.Source.TriggerEvent.TimeStamp))-
               MaxEdge.Tau);
--            Put(" A:");
--             Put(Integer(MaxEdge.DeltaT*100.0),3);
--             Put(Integer(MaxEdge.Sigma*100.0),3);
--             Put(Integer(MaxEdge.Weight*10.0),3);
         end if;
         -- NEVER forget to release a list handle!!
         EdgeList.Sec.Release(EdgeListHandle);

         -- go through all output edges (don't panic, just some minor things...)
         EdgeListHandle:=EdgeList.Sec.CreateHandle(Rec.OutputEdges);
         EdgeList.Sec.JumpToHead(EdgeListHandle);
         while EdgeList.Sec.IsValid(EdgeListHandle) loop
            ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
            -- forward propagation of time warp factor
            if Rec.Activity>LOW_ACT then
               ActEdge.Tau:=NewTau;
            else
               ActEdge.Tau:=1.0;
            end if;
            -- prophylactic decrease of edge weight
            ActEdge.Weight:=ActEdge.Weight-
              Rec.Stimulation*ActEdge.Lambda*C_D*ActEdge.Weight;
            -- and that's it!
            EdgeList.Sec.StepForward(EdgeListHandle);
         end loop;
         -- NEVER forget to release a list handle!!
         EdgeList.Sec.Release(EdgeListHandle);

         --Put("t: ");
         --Put(Integer(NewTau*100.0),3);
      exception

         when E: others =>
            Put_line("Error in DoReceptorCalculation!");
            Put_Line (Current_Error,
                      "Task "
                      & Image (Current_Task)
                      & " reports: "
                   & Exception_Name (E)
                      & " - "
                      & Exception_Message (E));
      end DoReceptorCalculation;


      procedure DoReceptorAdaptation(Rec:ReceptorAccess) is
         EdgeHandle:EdgeList.ListHandle;
         CrossEdgeListHandle:CrossEdgeList.ListHandle;
         ActEdge:EdgeAccess;
         EdgeActivity:NormRange;

         procedure DoAdaptation(EdgeListHandle:in out EdgeList.ListHandle) is
         begin
            -- go through all input edges

            EdgeList.Sec.JumpToHead(EdgeListHandle);

            while EdgeList.Sec.IsValid(EdgeListHandle) loop
               ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
               EdgeActivity:=CalcEdgeActivity(ActEdge.all);
               EdgeList.Sec.StepForward(EdgeListHandle);

               ActEdge.Sigma:=ActEdge.Sigma+ActEdge.Lambda*Rec.Stimulation*EdgeActivity*
                 (abs(TimeInterval(To_Duration(ActEdge.Target.TriggerEvent.TimeStamp-
                                              ActEdge.Source.TriggerEvent.TimeStamp))
                      -ActEdge.DeltaT)+ActEdge.DeltaT*MAX_TEMP_PRECISION-ActEdge.Sigma);

               ActEdge.DeltaT:=ActEdge.DeltaT+ActEdge.Lambda*Rec.Stimulation*EdgeActivity*
                 (ActEdge.Tau*
                  TimeInterval(To_Duration(ActEdge.Target.TriggerEvent.TimeStamp-
                                          ActEdge.Source.TriggerEvent.TimeStamp))
                  -ActEdge.DeltaT);

               ActEdge.Weight:=ActEdge.Weight+ActEdge.Lambda*Rec.Stimulation*EdgeActivity*
                 (1.0-ActEdge.Weight);

               ActEdge.Lambda:=ActEdge.Lambda-Rec.Stimulation*EdgeActivity*C_LAMBDA*ActEdge.Lambda;
            end loop;
            EdgeList.Sec.Release(EdgeListHandle);
         end;

      begin
         EdgeHandle:=EdgeList.Sec.CreateHandle(Rec.InputEdges);
         DoAdaptation(EdgeHandle);
         -- go through all cross edge lists

         CrossEdgeListHandle:=CrossEdgeList.Sec.CreateHandle(Rec.InCrossEdges);
         CrossEdgeList.Sec.JumpToHead(CrossEdgeListHandle);
         while CrossEdgeList.Sec.IsValid(CrossEdgeListHandle) loop
            EdgeHandle:=EdgeList.Sec.
```

```
        CreateHandle(CrossEdgeList.Sec.GetCurrent(CrossEdgeListHandle).Edges);
      DoAdaptation(EdgeHandle);
      CrossEdgeList.Sec.StepForward(CrossEdgeListHandle);
   end loop;
   CrossEdgeList.Sec.Release(CrossEdgeListHandle);
end DoReceptorAdaptation;


procedure SaveGraph(File:in File_Type) is
   EventListHandle:EventList.ListHandle;
   RecListHandle:ReceptorList.ListHandle;
   ActEventDescriptor:EventDescriptor;
   ActReceptor:ReceptorAccess;
   EdgeListHandle:EdgeList.ListHandle;
   ActEdge:EdgeAccess;
begin
   -- go through all events
   EventListHandle:=EventList.Sec.CreateHandle(MyEventList);
   EventList.Sec.JumpToHead(EventListHandle);
   while EventList.Sec.IsValid(EventListHandle) loop
      ActEventDescriptor:=EventList.Sec.GetCurrent(EventListHandle);
      -- and save all receptors
      RecListHandle:= ReceptorList.Sec.CreateHandle(ActEventDescriptor.CorrReceptors);
      ReceptorList.Sec.JumpToHead(RecListHandle);
      while ReceptorList.Sec.IsValid(RecListHandle) loop
         ActReceptor:=ReceptorList.Sec.GetCurrent(RecListHandle);
         Put_Line(File, "<Receptor>");
         TagThat(File, "ID",ActReceptor.ID);
         TagThat(File, "SeqID",ActReceptor.ID);
         TagThat(File, "EventClass", ActEventDescriptor.EventClass);
         TagThat(File, "PosX", ActReceptor.TriggerEvent.PosX);
         TagThat(File, "PosY", ActReceptor.TriggerEvent.PosY);
         -- go through all input edges
         EdgeListHandle:=EdgeList.Sec.CreateHandle(ActReceptor.InputEdges);
         EdgeList.Sec.JumpToHead(EdgeListHandle);
         Put_Line(File,"");
         while EdgeList.Sec.IsValid(EdgeListHandle) loop
            ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
            Put(File,"<InEdge>");
            TagThat(File, "Source",ActEdge.Source.ID);
            TagThat(File, "Target",ActEdge.Target.ID);
            TagThat(File, "Weight",Float(ActEdge.Weight), 1, 3);
            TagThat(File, "Lambda",Float(ActEdge.Lambda), 1, 3);
            TagThat(File, "DeltaT",Float(ActEdge.DeltaT), 4, 3);
            TagThat(File, "Sigma", Float(ActEdge.Sigma), 4, 3);
            Put_Line(File, "</InEdge>");
            EdgeList.Sec.StepForward(EdgeListHandle);
         end loop;
         EdgeList.Sec.Release(EdgeListHandle);
         Put_Line(File, "</Receptor>");
         ReceptorList.Sec.StepForward(RecListHandle);
      end loop;
      ReceptorList.Sec.Release(RecListHandle);
      EventList.Sec.StepForward(EventListHandle);
   end loop;
   EventList.Sec.Release(EventListHandle);
end SaveGraph;


procedure SaveAsFilterLayout(File:in File_Type) is
   EventListHandle:EventList.ListHandle;
   RecListHandle:ReceptorList.ListHandle;
   ActEventDescriptor:EventDescriptor;
   ActReceptor:ReceptorAccess;
   EdgeListHandle:EdgeList.ListHandle;
   ActEdge:EdgeAccess;
begin
   -- go through all events
   EventListHandle:=EventList.Sec.CreateHandle(MyEventList);
   EventList.Sec.JumpToHead(EventListHandle);
   while EventList.Sec.IsValid(EventListHandle) loop
      ActEventDescriptor:=EventList.Sec.GetCurrent(EventListHandle);
      -- and save all receptors
      RecListHandle:= ReceptorList.Sec.CreateHandle(ActEventDescriptor.CorrReceptors);
      ReceptorList.Sec.JumpToHead(RecListHandle);
      while ReceptorList.Sec.IsValid(RecListHandle) loop
         ActReceptor:=ReceptorList.Sec.GetCurrent(RecListHandle);
         Put(File, "<filter>");
         TagThat(File, "name","Receptor");
         -- go through all input edges
         EdgeListHandle:=EdgeList.Sec.CreateHandle(ActReceptor.OutputEdges);
         Put_Line(File,"");
         Put(File,ActReceptor.TriggerEvent.PosX,0,5,0);
         Put_Line(File,"");
         Put(File,ActReceptor.TriggerEvent.PosY,0,5,0);
         Put_Line(File,"");
         Put(File, "<ID>");
         Put(File, ActReceptor.ID,0);
```

```
            Put_Line(File,"");
            Put_Line(File,"<paramNo>4");
            TagThat(File, "description", "No. of Outputs");TagThat(File,"type",0,0);
            TagThat(File, "content", Long_Integer(EdgeList.Sec.Size(EdgeListHandle)),0);
            Put_Line(File,"");
            TagThat(File, "description", "ID");TagThat(File,"type",0,0);
            TagThat(File, "content",ActReceptor.ID,0);
            Put_Line(File,"");
            TagThat(File, "description", "SeqID");TagThat(File,"type",0,0);
            TagThat(File, "content",ActReceptor.SeqID,0);
            Put_Line(File,"");
            TagThat(File, "description", "EventClass");TagThat(File,"type",0,0);
            TagThat(File, "content", ActEventDescriptor.EventClass,0);

            EdgeList.Sec.Release(EdgeListHandle);
            Put_Line(File,"");
            Put_Line(File, "</filter>");
            ReceptorList.Sec.StepForward(RecListHandle);
         end loop;
         ReceptorList.Sec.Release(RecListHandle);
         EventList.Sec.StepForward(EventListHandle);
      end loop;

      Put_Line(File, "<connections>");

      EventList.Sec.JumpToHead(EventListHandle);
      while EventList.Sec.IsValid(EventListHandle) loop
         ActEventDescriptor:=EventList.Sec.GetCurrent(EventListHandle);
         -- and save all receptors
         RecListHandle:= ReceptorList.Sec.CreateHandle(ActEventDescriptor.CorrReceptors);
         ReceptorList.Sec.JumpToHead(RecListHandle);
         while ReceptorList.Sec.IsValid(RecListHandle) loop
            ActReceptor:=ReceptorList.Sec.GetCurrent(RecListHandle);
            -- go through all output edges
            EdgeListHandle:=EdgeList.Sec.CreateHandle(ActReceptor.OutputEdges);
            EdgeList.Sec.JumpToHead(EdgeListHandle);
            while EdgeList.Sec.IsValid(EdgeListHandle) loop
               ActEdge:=EdgeList.Sec.GetCurrent(EdgeListHandle);
                  Put(File, ActEdge.Target.ID,0);
                  Put_Line(File, "");
                  Put(File, Integer(ActEdge.Weight*100.0),0);
                  Put_Line(File, "");
               EdgeList.Sec.StepForward(EdgeListHandle);
            end loop;
            EdgeList.Sec.Release(EdgeListHandle);
            ReceptorList.Sec.StepForward(RecListHandle);
         end loop;
         ReceptorList.Sec.Release(RecListHandle);
         EventList.Sec.StepForward(EventListHandle);
      end loop;


      EventList.Sec.Release(EventListHandle);
   end SaveAsFilterLayout;



end Sec;



procedure Save is
   File:File_Type;
begin
   -- open File
   Create(File, Name=>"graph.top");
   Sec.SaveGraph(File);
   Close(File);
   Create(File, Name=>"graph.sys");
   Sec.SaveAsFilterLayout(File);
   Close(File);
end;


task body InputCalculations is

   LastCreatedReceptor, NewCreatedReceptor:ReceptorAccess;

   function FindEventDescriptor(ForEvent:Event) return EventDescriptor is
      EventListHandle:EventList.ListHandle;
      Found:Boolean;
      LastEventDescriptor:EventDescriptor;
   begin
      EventListHandle:=EventList.Sec.CreateHandle(MyEventList);
      -- check if event is already known:
      EventList.Sec.JumpToHead(EventListHandle);
      Found:=False;
      while EventList.Sec.IsValid(EventListHandle) and not found loop
```

```
            LastEventDescriptor:=EventList.Sec.GetCurrent(EventListHandle);
            if LastEventDescriptor.EventClass=ForEvent.EventClass then
               Found:=True;
            end if;
            EventList.Sec.StepForward(EventListHandle);
         end loop;
         EventList.Sec.Release(EventListHandle);
         if not Found then
            -- insert new event into list
            LastEventDescriptor.EventClass:=ForEvent.EventClass;
            LastEventDescriptor.CorrReceptors:=ReceptorList.Sec.CreateList;
            EventList.Sec.Add(EventListHandle, LastEventDescriptor);
         end if;
         return LastEventDescriptor;
      exception
         when others => null;
            return LastEventDescriptor;
      end FindEventDescriptor;


      -- calculate HypothesisActivity
      -- it's increased, when the most active receptor of the preceeding event
      -- has a connection to the most active receptor of this event
      -- (continuity is rewarded)
      -- if there is no connection, an edge is created

      procedure ProvideHypothesisContinuity is
         EdgeHandle:EdgeList.ListHandle;
         Continuity:Boolean:=false;
         NewEdge:EdgeAccess;
      begin

         if OldActivityList(1)/=null and ActivityList(1)/=null then
            EdgeHandle:=EdgeList.Sec.CreateHandle(OldActivityList(1).OutputEdges);
            EdgeList.Sec.JumpToHead(EdgeHandle);
            while EdgeList.Sec.IsValid(EdgeHandle) loop
               -- if there is a connection...
               if EdgeList.Sec.GetCurrent(EdgeHandle).Target.Id=ActivityList(1).ID then
                  Continuity:=True;
                  EdgeList.Sec.Release(EdgeHandle);
               else
                  EdgeList.Sec.StepForward(EdgeHandle);
               end if;
            end loop;
            EdgeList.Sec.Release(EdgeHandle);
            if Continuity then
               HypothesisActivity:=(1.0+HypothesisActivity)/2.0;
            else
               HypothesisActivity:=(0.0+HypothesisActivity)/2.0;
               if OldActivityList(1)/=null and then ActivityList(1)/=null
                 and then OldActivityList(1).Activity>HIGH_ACT
                 and then ActivityList(1).Activity>HIGH_ACT
               then
                  NewEdge:=CreateEdge(OldActivityList(1), ActivityList(1));
                  NewEdge.Weight:=ActivityList(1).Activity;
               end if;
            end if;
         else
            HypothesisActivity:=(0.0+HypothesisActivity)/2.0;
         end if;
      end;

      -- local variables of InputCalculations

      LastEvent:Event;
      LastEventDescriptor:EventDescriptor;
      RecListHandle:ReceptorList.ListHandle;
      ActReceptor:ReceptorAccess;
      Index:Integer;
      ActEdge:EdgeAccess;

      ConcHandle:ModalityList.ListHandle;

      TransActList:TransportActivityList;
      ActMessage:Message;
      ModDescriptor:ModalityDescriptor;

   begin -- of InputCalculations
      Put_Line("Local mapping active.");
      while ProgramActive loop
         -- wait for new event
         InputHandler.GetEvent(LastEvent);
         DreamTask.Suspend;
         if ProgramActive then
            Put(" Event:  ");
            Put(Integer(LastEvent.EventClass),3);

            LastEventDescriptor:=FindEventDescriptor(LastEvent);
            -- get list of corresponding receptors
            RecListHandle:= ReceptorList.Sec.CreateHandle(LastEventDescriptor.CorrReceptors);
            ReceptorList.Sec.JumpToHead(RecListHandle);

            Sec.ClearActivityList;
            -- and do calcs for every single one of them...
            while ReceptorList.Sec.IsValid(RecListHandle) loop
               ActReceptor:=ReceptorList.Sec.GetCurrent(RecListHandle);
               ActReceptor.Stimulation:=LastEvent.Similarity;
               ActReceptor.TriggerEvent.TimeStamp:=LastEvent.TimeStamp;
               ActReceptor.TriggerEvent.Similarity:=LastEvent.Similarity;
               Sec.DoReceptorCalculation(ActReceptor);

               -- maintain activity list
               Sec.Add2ActivityList(ActReceptor);

               ReceptorList.Sec.StepForward(RecListHandle);
            end loop;

            -- adaptation of most active receptor
            if ActivityList(1)/=null then
               Sec.DoReceptorAdaptation(ActivityList(1));
            end if;

            ProvideHypothesisContinuity;
            PrintActivityList;

            if LastEvent.Similarity>HIGH_ACT then
               -- go through ActivityList
--               for I in 1..MaxConcHypotheses loop
                  -- create edges from most recently inserted receptor to emerging waves
                  if LastCreatedReceptor/=null
                    and then ActivityList(1)/=null
                    and then ActivityList(1).Activity>=HIGH_ACT then
                     Put(" CL:");
                     Put(Integer(LastCreatedReceptor.TriggerEvent.EventClass),2);
                     Put(Integer(ActivityList(1).TriggerEvent.EventClass),2);
                     ActEdge:=CreateEdge(LastCreatedReceptor, ActivityList(1));
                  end if;
--               end loop;

               -- do acquisition, if necessary
               if ActivityList(1)=null or else ActivityList(1).Activity<HIGH_ACT then
                  --Put(" Create Receptor..");
                  if OldActivityList(1)/=null
                    and then OldActivityList(1).Activity>HIGH_ACT then
                     CurrentSeqID:=CurrentSeqID+1;
                  end if;
                  NewCreatedReceptor:=CreateReceptor(LastEventDescriptor);
                  NewCreatedReceptor.Stimulation:=LastEvent.Similarity;
                  NewCreatedReceptor.TriggerEvent:=LastEvent;
                  NewCreatedReceptor.SeqID:=CurrentSeqID;
                  Put(" NR");
                  Put(Integer(NewCreatedReceptor.ID),3);

                  if LastCreatedReceptor/=null then
                     ActEdge:=CreateEdge(LastCreatedReceptor, NewCreatedReceptor);
                     Put(" LN:");
                     Put(Integer(LastCreatedReceptor.TriggerEvent.EventClass),2);
                     Put(Integer(NewCreatedReceptor.TriggerEvent.EventClass),2);
                  end if;
--                  for I in 1..MaxConcHypotheses loop
--if OldActivityList(I)/=null
                  if OldActivityList(1)/=null
                     and then OldActivityList(1).Activity>=HIGH_ACT
                     then
                        ActEdge:=CreateEdge(OldActivityList(1), NewCreatedReceptor);
                        Put(" LO:");
                        Put(Integer(OldActivityList(1).TriggerEvent.EventClass),2);
                        Put(Integer(NewCreatedReceptor.TriggerEvent.EventClass),2);
                     end if;
--                  end loop;
                  LastCreatedReceptor:=NewCreatedReceptor;
               else
                  LastCreatedReceptor:=null;
               end if;
               ReceptorList.Sec.Release(RecListHandle);
            end if;

            -- ********************************************************************************
            -- sensor fusion part
            -- creation of cross-edges

            if ActivityList(1)/=null then
               ConcHandle:=ModalityList.Sec.CreateHandle(ConcurrentActivityList);
               ModalityList.Sec.JumpToHead(ConcHandle);
               while Modalitylist.Sec.IsValid(ConcHandle) loop
                  ModDescriptor:=ModalityList.Sec.GetCurrent(ConcHandle);
                  if ModDescriptor.LastMostActive/=null then
                     ActEdge:=CreateEdge(ModDescriptor.LastMostActive, ActivityList(1));
                     if ActEdge/=null then
                        Put("CE "&NumID'Image(ModDescriptor.LastMostActive.ID));
                     end if;
                     ModDescriptor.LastMostActive:=null;
                     ModalityList.Sec.OverwriteElement(ConcHandle, ModDescriptor);
                  end if;
                  ModalityList.Sec.StepForward(ConcHandle);
               end loop;
               ModalityList.Sec.Release(ConcHandle);
            end if;


            -- send activity to other receptors
            for I in 1..MaxSubmitHypotheses loop
               if ActivityList(I)/=null then
                  TransActList(I).Activity:=ActivityList(I).Activity;
                  TransActList(I).TimeStamp:=To_Duration(ActivityList(I).TriggerEvent.TimeStamp-Time_First);
                  TransActList(I).InvSensors:=ActivityList(I).InvSensors;
                  TransActList(I).ID:=ActivityList(I).ID;
                  TransActList(I).SeqID:=ActivityList(I).SeqID;
               else
                  TransActList(I).Activity:=-1.0;
               end if;
            end loop;
            ActMessage.Content.ActiveReceptors:=TransActList;
            LocalBroadCast(ActMessage);


            Put_Line("");
            -- activate anticipation task
            DreamTask.Notify;

         end if;
      end loop;
      ShutDownCommunication;
      Put_Line("Local mapping shut down.");
   exception
      when E: others =>
         Put_Line("Error occurred in InputCalculations!");
         Put_Line (Current_Error,
                   "Task "
                   & Image (Current_Task)
                   & " reports: "
                   & Exception_Name (E)
                   & " - "
                   & Exception_Message (E));
   end InputCalculations;



   task body DreamTask is
      WakeUpTime:Time;
      ActReceptor,MinReceptor:ReceptorAccess;
      ActEdge, MinEdge:EdgeAccess;
      EHandle:EdgeList.ListHandle;
      ImaginatedEvent:Event;
      Suspended, AlreadyHappened:Boolean:=True;

      DreamList:ReceptorList.ListDescriptor;

      procedure CalcWakeUp is
         Now:Time;
         DreamHandle, TmpHandle:ReceptorList.ListHandle;
         Index:Integer;
      begin
         Now:=GetGlobalTime;
         -- find next anticipated event
         MinReceptor:=null;
         MinEdge:=null;
         DreamHandle:=ReceptorList.Sec.CreateHandle(DreamList);
         ReceptorList.Sec.JumpToHead(DreamHandle);
         Index:=1;
         while Index<=MaxConcHypotheses or ReceptorList.Sec.IsValid(DreamHandle) loop
            if Index<=MaxConcHypotheses then
               ActReceptor:=ActivityList(Index);
               Index:=Index+1;
            else
               ActReceptor:=ReceptorList.Sec.GetCurrent(DreamHandle);
               TmpHandle:=ReceptorList.Sec.DuplicateHandle(DreamHandle);
               ReceptorList.Sec.StepForward(DreamHandle);
            end if;
            if ActReceptor/=null then
               EHandle:=EdgeList.Sec.CreateHandle(ActReceptor.OutputEdges);
               EdgeList.Sec.JumpToHead(EHandle);

               while EdgeList.Sec.IsValid(EHandle) loop
```

```
                  ActEdge:=EdgeList.Sec.GetCurrent(EHandle);
                  if ActEdge/=null
                    and then ActReceptor.Activity>LOW_ACT
                    and then (ActReceptor.TriggerEvent.TimeStamp+
                         To_Time_Span(Duration(ActEdge.DeltaT)))>Now
                    and then
                    (MinReceptor=null or else
                      (MinReceptor/=null and then MinEdge/=null
                      and then (ActReceptor.TriggerEvent.TimeStamp+
                       To_Time_Span(Duration(ActEdge.DeltaT)))<
                      (MinReceptor.TriggerEvent.TimeStamp+
                       To_Time_Span(Duration(MinEdge.DeltaT)))
                      and then
                      (ActReceptor.TriggerEvent.TimeStamp+
                       To_Time_Span(Duration(ActEdge.DeltaT)))>
                        InternalLastEvent.TimeStamp))
                  then
                     AlreadyHappened:=False;
                     for I in 1..MaxConcHypotheses loop
                        if ActEdge.Target=ActivityList(I) then AlreadyHappened:=True; end if;
                     end loop;
                     if not AlreadyHappened then
                        MinReceptor:=ActReceptor;
                        MinEdge:=ActEdge;
                     end if;
                  end if;
                  EdgeList.Sec.StepForward(EHandle);
               end loop;

               -- if event expired, delete it from Activity list
               if MinEdge=null then
                  ReceptorList.Sec.Remove(TmpHandle);
               end if;
               ReceptorList.Sec.Release(TmpHandle);
            end if;
         end loop;
         if MinReceptor/=null and MinEdge/=null then
            WakeUpTime:=MinReceptor.TriggerEvent.TimeStamp+To_Time_Span(Duration(MinEdge.DeltaT));
            Suspended:=False;
         else
            WakeUpTime:=GetGlobalTime+To_Time_Span(Duration(10.0));
            Suspended:=True;
         end if;
         ReceptorList.Sec.Release(DreamHandle);

      end CalcWakeUp;

      DreamHandle, TmpHandle:ReceptorList.ListHandle;

   begin
      DreamList:=ReceptorList.Sec.CreateList;
      WakeUpTime:=GetGlobalTime+To_Time_Span(Duration(10.0));
      MinEdge:=null;
      MinReceptor:=null;
      Suspended:=True;
      while ProgramActive loop
         select
            accept Suspend do
               Suspended:=True;
            end Suspend;
         or
            accept Notify do
               -- include current activity list
               --DreamHandle:=ReceptorList.Sec.CreateHandle(DreamList);

               --for I in 1..MaxConcHypotheses loop
               --   if ActivityList(I)/=null then
               --      ReceptorList.Sec.GoToElement(DreamHandle, OldActivityList(I));
               --      if ReceptorList.Sec.IsValid(DreamHandle) then
               --         TmpHandle:=ReceptorList.Sec.DuplicateHandle(DreamHandle);
               --         ReceptorList.Sec.Remove(TmpHandle);
               --      end if;
               --      ReceptorList.Sec.GoToElement(DreamHandle, ActivityList(I));
               --      if not ReceptorList.Sec.IsValid(DreamHandle) then
               --         ReceptorList.Sec.Add(DreamHandle, ActivityList(I));
               --      end if;
               --   end if;
               --end loop;
               --ReceptorList.Sec.Release(DreamHandle);
               -- and determine next wakeup time
               CalcWakeUp;
               Suspended:=False;
            end Notify;
         or
            delay until WakeUpTime;
            if not Suspended and MinEdge/=null then
               ActReceptor:=MinEdge.Target;
               -- only trigger if event has not already happened!
               if ActReceptor.TriggerEvent.TimeStamp<WakeUpTime then
                  ActReceptor.TriggerEvent.TimeStamp:=WakeUpTime;
                  ActReceptor.TriggerEvent.Similarity:=1.0/((1.0+C_DAMP)*(1.0+C_A));
                  ActReceptor.Stimulation:=ActReceptor.TriggerEvent.Similarity;
                  Sec.CalcReceptorActivity(ActReceptor);
                  --                 Put(" Dream:");
--                   Put(Integer(ActReceptor.TriggerEvent.EventClass),2);
                  --               Put("  Activity:");
                  --                Put(Integer(100.0*MinReceptor.Activity),3);
                  --               Put("%");
                  --               PrintActivityList;
                  if ActReceptor.Activity>LOW_ACT then
                     DreamHandle:=ReceptorList.Sec.CreateHandle(DreamList);
                     ReceptorList.Sec.Add(DreamHandle,ActReceptor);
                     ReceptorList.Sec.Release(DreamHandle);
                  end if;
                  CalcWakeUp;
  --               Put_Line("");
               end if;
            else
               WakeUpTime:=GetGlobalTime+To_Time_Span(Duration(10.0));
            end if;
         end select;
      end loop;
   exception
      when E: others =>
         Put_Line("Error occurred in DreamTask!");
         Put_Line (Current_Error,
                   "Task "
                   & Image (Current_Task)
                   & " reports: "
                   & Exception_Name (E)
                   & " - "
                   & Exception_Message (E));

   end DreamTask;


   task body ConcurrentActivityHandler is
      ActMsg:Message;
      SenderID:NumID;
      ModHandle:ModalityList.ListHandle;
      RecHandle, ConcHandle:ReceptorList.ListHandle;
      ModDescriptor:ModalityDescriptor;
      ActReceptor:ReceptorAccess;
   begin
      ConcurrentActivityList:=ModalityList.Sec.CreateList;
      ConcurrentActivityTaskReady:=True;
      while ProgramActive loop
         PrivateMonitor.Monitor.Listen(SenderID,ActMsg);
         -- find corresponding descriptor
         ModHandle:=ModalityList.Sec.CreateHandle(ConcurrentActivityList);
         ModalityList.Sec.JumpToHead(ModHandle);
         while ModalityList.Sec.IsValid(ModHandle)
               and then ModalityList.Sec.GetCurrent(ModHandle).ModID/=SenderID loop
            ModalityList.Sec.StepForward(ModHandle);
         end loop;
         if ModalityList.Sec.IsValid(ModHandle) then
            ModDescriptor:=ModalityList.Sec.GetCurrent(ModHandle);
         else
            ModDescriptor.ModID:=SenderID;
            ModDescriptor.Receptors:=ReceptorList.Sec.CreateList;
            ModDescriptor.ClockSkew:=Duration(0.0);
            ModalityList.Sec.Add(ModHandle, ModDescriptor);
         end if;

         -- update activity of existing cross edge sources
         RecHandle:=ReceptorList.Sec.CreateHandle(ModDescriptor.Receptors);

         for I in 1..MaxSubmitHypotheses loop
            -- identify Receptor
            if  ActMsg.Content.ActiveReceptors(I).Activity>0.0 then
               ReceptorList.Sec.JumpToHead(RecHandle);
               while ReceptorList.Sec.IsValid(RecHandle)
                 and then ReceptorList.Sec.GetCurrent(RecHandle).ID/=ActMsg.Content.ActiveReceptors(I).ID loop
                  ReceptorList.Sec.StepForward(RecHandle);
               end loop;
               -- if receptor could be identified, update data
               if ReceptorList.Sec.IsValid(RecHandle) then
                  ActReceptor:=ReceptorList.Sec.GetCurrent(RecHandle);
                  -- update parameters
                  ActReceptor.Activity:=ActMsg.Content.ActiveReceptors(I).Activity;
                  --ActReceptor.TriggerEvent.TimeStamp:=Time_First+
                  --  To_Time_Span(ModDescriptor.ClockSkew+ActMsg.Content.ActiveReceptors(I).TimeStamp);
                  ActReceptor.TriggerEvent.TimeStamp:=GetGlobalTime;

               else
                  -- otherwise, create new CrossMirror receptor
                  ActReceptor:=GetRecycledReceptor;
                  ActReceptor.CrossMirror:=True;
                  ActReceptor.ID:=ActMsg.Content.ActiveReceptors(I).ID;
                  ActReceptor.Activity:=ActMsg.Content.ActiveReceptors(I).Activity;
                  -- global time calculation!!!!
                  --ActReceptor.TriggerEvent.TimeStamp:=Time_First+
                  --  To_Time_Span(ModDescriptor.ClockSkew+ActMsg.Content.ActiveReceptors(I).TimeStamp);
                  ActReceptor.TriggerEvent.TimeStamp:=GetGlobalTime;
                  ReceptorList.Sec.Add(RecHandle, ActReceptor);
               end if;

               -- write receptor in concurrent activity cache, if it's a new maximum...
               if ModDescriptor.LastMostActive=null
                 or else ActReceptor.Activity>ModDescriptor.LastMostActive.Activity then
                  ModDescriptor.LastMostActive:=ActReceptor;
                  ModalityList.Sec.OverwriteElement(ModHandle, ModDescriptor);
               end if;
            end if;
         end loop;
         ModalityList.Sec.Release(ModHandle);
         ReceptorList.Sec.Release(RecHandle);
      end loop;
   end ConcurrentActivityHandler;


begin
   ReceptorGarbage:=ReceptorList.Sec.CreateList;
   EdgeGarbage:=EdgeList.Sec.CreateList;
   MyEventList:=EventList.Sec.CreateList;

   Text_IO.Put_Line("StartUp...");
   if Argument_Count>=2 then
      SetInternalAdress(Argument(1), Integer'Value(Argument(2)));
      if Argument_Count>=4 then
         Register(Argument(3), Integer'Value(Argument(4)));
      end if;
   else
      Put_Line("Arguments missing: <LocalIP> <ListenPort> [<PartitionServerIP> <PartitionServerPort>]");
   end if;


end GraphManager;
```

Bibliography

[1] Valentin Braitenberg, Detlef Heck, and Fahad Sultan. The detection and generation of sequences as a key to cerebellar function: Experiments and theory. *Behavioral and Brain Sciences*, 20:229–277, 1997.

[2] G. de A. Barreto and A. F. R. Araújo. Time in self-organizing maps: An overview of models. *International Journal of Computer Research*, 10(2):139–179, 2001.

[3] G. de A. Barreto, A. F. R. Araújo, C. Dücker, and H. Ritter. Implementation of a distributed robotic control system based on a temporal self-organizing network. In *Proc. of the IEEE Int. Conf. on Syst., Man, and Cybern. (SMC'01)*, pages 335–340, Tucson, Arizona, 2001.

[4] N. R. Euliano and J. C. Principe. A spatio-temporal memory based on SOMs with activity diffusion. In S. Oja, E. & Kaski, editor, *Kohonen Maps*, pages 253–266. Elsevier, Amsterdam, 1999.

[5] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem.

[6] Stefan B. Williams, Gamini Dissanayake, and Hugh Durrant-Whyte. Towards multi-vehicle simultaneous localisation and mapping. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation Washington DC*, 2002.

[7] Uwe R. Zimmer. Embedding local metrical map patches in a globally consistent topological map. In *Proceedings of Underwater Technologies 2000, Tokyo, Japan*, pages 23–26, 2000.