Universität Leipzig
Fakultät für Mathematik und Informatik
(Institut für Informatik)

Simulation and Investigation of Embodied Autonomous Robots driven by Homeokinetic Control

Diplomarbeit

vorgelegt von: Robert Pies

Leipzig, im April 2001

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, all jenen meinen Dank auszusprechen, die mich in meiner Arbeit unterstützt und gefördert haben und somit nicht unwesentlich an ihrer Entstehung beteiligt sind:

Herrn Prof. Dr. Ralf Der welcher diese Arbeit betreut hat und dem ich für die stets sehr anregenden Gespräche danke, wobei er mir mit bestem Rat zur Seite stand, immer ein offenes Ohr für mich hatte und mir überdies die Freiheit ließ, die ich zur Erstellung dieser Arbeit benötigte.

Meinen Eltern und Reinhardt für die Unterstützung während der Entstehung dieser Arbeit und bei meinem Studium.

Sowie, last but not least, Christian für die gute Freundschaft und für die vielen erfolgreichen Aufheiterungen in Phasen der intensiven Arbeit.

Abstract

The thesis is concerned with modeling and simulation of autonomous robots, embodied in dynamic environments, in order to investigate emergent behavior of situated agents interacting with their environment.
Hereby modeling refers to capturing the essentials of real-world physics for simple objects and their interactions within the world as well as modeling the internal mechanisms driving the robots.
Thus, simple models of the object's properties, dynamics and their interaction are introduced first. These formulations are then integrated into a coherent framework suited for implementation and execution on a computer. Emphasis is put on simple and less detailed (prototypical) types of objects (simple robots, balls and static objects) yet with unrestricted and precise interaction, rather than focusing on static scenes consisting of specific complex real-world robots and objects.
The remaining part deals with modeling robot-behavior or rather a framework suited for emergence of behavior, namely homeokinesis. Experiments are conducted at this stage, investigating the applicability of the framework to specific settings leading to insight into the notion of homeokinesis and demonstrating its power as well as its limitations.
Finally, by application of recurrent neural-networks, remedy is provided for an encountered limitation of the original homeokinetic formulation.
The simulator developed in the first part is utilized in these experiments and investigations, proving to be a valuable and flexible tool.

Contents

List of Figures

List of Classes

Chapter 1

Introduction

The task of investigating embodied autonomous robots by means of simulation is undertaken in this thesis in the spirit of *synthetic modeling*, being just one of a set of keywords, such as *embodiment*, *emergence*, *situatedness* and others, which have become popular in Artificial Intelligence (AI), Artificial Life (AL) and related disciplines over the past years.

However, the relations to the work in those fields are not stressed here very much, although without having been exposed to this "modern" AI (as opposed to *classical* AI), I would hardly have written the same thesis.

An overview of the thesis is given by the following: Beginning with the second chapter, models will be derived for single objects which may be part of a simulated world. In chapter 3, these models are accompanied by collision-prediction techniques (this topic is only touched upon briefly) and means for handling collision-effects, modeling the interactions among multiple objects. In chapter 4 all models are integrated into a coherent framework enabling its *discrete-event-triggered* execution on a computer, thus providing an efficient implementation of a simulator.

Starting with chapter 5, the remaining part of the thesis is dedicated to investigating application of homeokinetic control to robots in the kinds of settings suited for simulation by the developed simulator. In chapter 6, some encountered limitations in the original homeokinetic formulation are remedied by utilizing *recurrent neural networks*.
Chapter 7 provides a conclusion for the thesis.

Chapter 2

Models of Single Physical Objects

Models for single objects in the world to be simulated will be derived in this chapter. Hereby it is stressed, that the world's structure is held rather simple and is meant to represent a *prototypical* environment for investigating autonomous robots, rather than simulating any specific type of robot. Emphasis is put on geometric and dynamic simplicity, but accurate dynamics and unrestricted interaction between (arbitrarily many) objects rather than on complex static scenes with little interaction or detailed models of single robots.

Therefore, the models to be simulated are also rather simplified and in some aspects represent abstracted versions of the behavior of real physical objects.
Major simplifications are the reduction of a true 3-dimensional world to a 2-dimensional representation as well as neglection of angular-friction effects in the dynamic models. As a justification for these simplifications it is stressed again, that the objects in focus are not to mimic any real-world counterparts in an exact manner, but are rather motivated by real objects and represent (prototypical) models of the *type* of object they abstract.
Another major abstraction is the simulator's possibility to apply periodic boundary conditions to physical space in the simulation (that is to say the worlds borders may be *wrapped around*), which can be quite useful for avoiding finite size effects and simulating worlds of infinite space.

Static Objects

The only type of static object considered is any polygonal object in two dimensions.

Dynamic Objects

Although the simulator to be presented is not aimed at simulating a special kind of existing robot, its robot-model is strongly oriented at the Khepera robot (see Figure 2.1) which is widely used in autonomous robotics research[1]. Besides robots, the only other dynamic objects are balls which may be rolling when hit by other objects, but are completely passive otherwise.

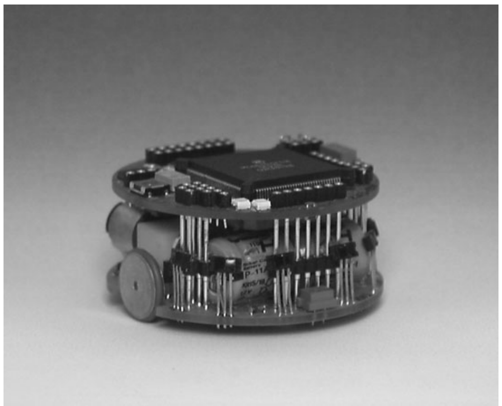


Figure 2.1: Picture of the Khepera autonomous mobile robot (© by K-Team, Switzerland), equipped with eight infrared-proximity and eight light sensors for perception.
Picture with courtesy of Francesco Mondada.

Properties of Objects and the Notion of State

All of the three basic classes of objects have specific properties, such as geometric properties (all types of objects), momentum and mass (only dynamic objects) or driving-speed (only robots).
A property can either be constant (e.g. mass, geometric properties) or varying over time (e.g. speed, position).
For the varying values, dynamic models need to be designed in order to allow simulation of their evolution over time.

The set of all properties of an object (constant or variable) is called its *state*. In a given world the *overall state* (or *total state*), containing the state of each object in that world, includes all information necessary in order to determine future states of all objects in that world.
In the models, all shapes of dynamic object's can be reduced to 2 dimensions by simply neglecting the third dimension (height). In the case of balls and (cylindrical) robots, their 2-dimensional simplifications simply result in circles whereas polygons are used for static objects.
In detail, the state of an object i is described by the following data, where some constants (such as constants of friction etc.) are suppressed:

[1]This choice is made not only due to personal experience with this type of robot, but it is also suited very well for simulation because of its geometric and dynamic simplicity.

- Static objects:
  - $\{\vec{p_i}\}$: a set of points, implying a polygonal shape.
- Balls:
  - $\vec{x}_i$: position in world-coordinates,
  - $\vec{v}_i$: velocity,
  - r_i : radius,
  - m_i : mass.
- Robots:
  - $\vec{x}_i$, θ_i : position and orientation,
  - v_i^s, ω_i : forward-speed and angular velocity of the robot,
  - r_i : radius,
  - m_i : mass,
  - various variables holding information about its sensors and
  - various variables needed by the controller to determine values for v_i^s and ω_i at each controller-step.

The naming convention introduced in this list will be used throughout the thesis, where index i may be omitted when superfluous.

2.1 Dynamic Models of the Objects

The dynamic model of an object in the world to be simulated serves the purpose of enabling the simulation-engine to keep track of the *state* of the object in question.
In particular the positions and velocities of all objects need to be tracked, whereas variables such as mass and radius of an object need to be known as well (and are therefore included in the concept state), but are not subject to change and are thus not included in the dynamic model.

2.1.1 Assumptions and Simplifications

The dynamic models for balls and robots will be derived from the following assumptions, observations and simplifications:

- Balls:
  - A ball experiences two kinds of forces:
    An impulsive collision-force, when it collides with another object (static or dynamic) and while it is moving (force of friction).
  - For velocities above a minimum v_{min}, a ball's experienced force due to friction is simplified to be linearly related to its velocity.
  - Below v_{min}, the balls velocity is assumed to be negligible and is therefore set to zero.
- Robots:
  - A robot also experiences two kinds of forces:
    One when it collides with other objects and a force due to its drive, consisting of two independently operating motors with turn-speeds v^l and v^r for the left and right wheel, respectively.
  - The two wheel-speeds v^l and v^r can be mapped to angular velocity ω and (forward-) speed v^s (which is proportional to the traveled distance per unit time) in a 1-to-1-fashion.
  - The robot's drive is set to a constant velocity during intervals between successive controller-steps yielding new velocities. (The robot's motors are hereby idealized and assumed to be capable of providing the desired velocities instantly - any inertial and slipping effects are neglected).

2.1.2 Dynamic Model of a Ball

The above stated assumptions and simplifications directly translate into the following model for a ball, when $\|\vec{v}(t)\| \geq v_{min}$:

$$\vec{a}(t) = -\mu_f \, \vec{v}(t), \tag{2.1.1}$$

where the object's index i is omitted for ease of notation and $\vec{a}(t)$, $\vec{v}(t)$ and μ_f denote acceleration, velocity and the friction-coefficient of the ball, respectively.
Since $\vec{a}(t) = \ddot{\vec{x}}(t)$ and $\vec{v}(t) = \dot{\vec{x}}(t)$ integration of $\vec{a}(t)$:

$$\vec{v}(t) = \int_{t_0}^{t} \vec{a}(\tau) \, d\tau \tag{2.1.2}$$

$$\vec{x}(t) = \int_{t_0}^{t} \vec{v}(\tau) \, d\tau \tag{2.1.3}$$

and initial conditions:

$$\vec{v}(t_0) = \vec{v}_0 \tag{2.1.4}$$
$$\vec{x}(t_0) = \vec{x}_0 \tag{2.1.5}$$

yield the desired dynamic model of a ball (in explicit form):

$$\vec{v}(t) = \vec{v}_0 \, e^{-\mu_f \triangle t} \tag{2.1.6}$$
$$\vec{x}(t) = \vec{x}_0 + \frac{\vec{v}_0}{\mu_f} \, (1 - e^{-\mu_f \triangle t}), \tag{2.1.7}$$

where $\vec{x}(t)$ denotes the position of the ball and $\triangle t = t - t_0$ denotes the time passed since the ball was at position $\vec{x}_0$, having velocity $\vec{v}_0$ (an illustrative example in one dimension is given in Figure 2.2).

It is stressed again, that the model given above is only valid for times $t \leq t_{pass}$, where $\|\vec{v}(t)\| = v_{min}$. For times $t > t_{pass}$, when the ball is passive, the model is trivial, since $\vec{v}(t) = \vec{0}$ and $\vec{x}(t) = \vec{x}(t)$. This raises the issue of determining t_{pass}, which is given by:

$$t_{pass} = \frac{1}{\mu} \ln \left(\frac{\|\vec{v}(t)\|}{v_{min}} \right) \tag{2.1.8}$$

The last equation is needed in Chapter 4, when all models are integrated into a coherent framework.

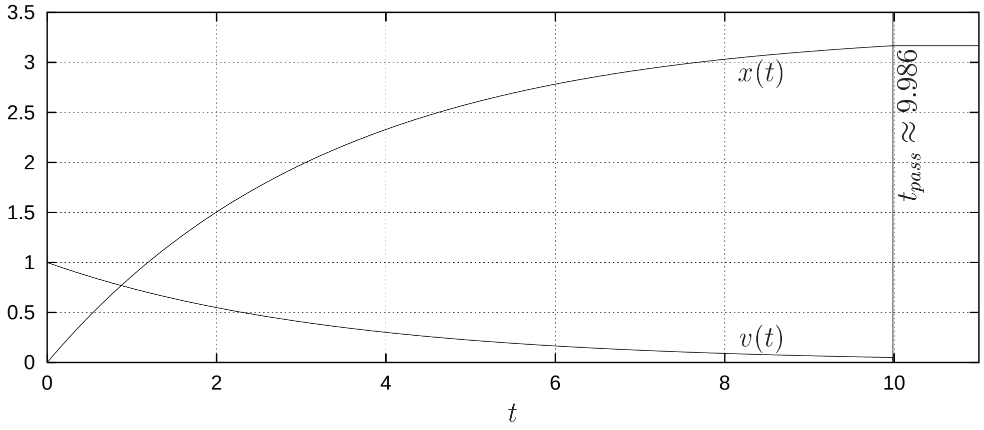


Figure 2.2: Illustration of the Dynamic model of a ball in one dimension: At time $t_0 = 0$, a ball starts with velocity $v_0 = 1$ at position $x_0 = 0$. The constants are: $\mu_f = 0.3$, $v_{min} = 0.05$ and thus, at time $t_{pass} \approx 9.986$ the ball stops rolling at $x(t_{pass}) \approx 3.167$.

2.1.3 Dynamic Model of a Robot

According to the simplifications and assumptions stated earlier, a robot's model can be derived by considering driving speed v^s and angular velocity ω (as opposed to acceleration in the case of a ball). In particular $\dot{\theta}(t) = \omega(t)$ and $\dot{\vec{x}}(t) = \vec{v}(t)$, thus the dynamic model can be derived by starting with the equations:

$$\theta(t) = \int_{t_0}^{t} \omega(\tau)\, d\tau \tag{2.1.9}$$

$$\vec{x}(t) = \int_{t_0}^{t} \vec{v}(\tau)\, d\tau, \tag{2.1.10}$$

where $\theta(t)$ and $\vec{x}(t)$ denote the position of the robot as described in the last section and initial conditions:

$$\theta(t_0) = \theta_0 \tag{2.1.11}$$

$$\vec{x}(t_0) = \vec{x}_0. \tag{2.1.12}$$

In Equation (2.1.10) $\vec{v}(t)$ denotes the robot's tangential velocity, which is given by:

$$\vec{v}(t) = v^s \begin{bmatrix} \cos(\theta(t)) \\ \sin(\theta(t)) \end{bmatrix} \tag{2.1.13}$$

Since ω (as well as v^s) is constant in intervals between successive controller-steps t_k and t_{k+1} (for $t \in [t_k, t_{k+1})$, $\omega(t) = \omega_k$ is just the angular velocity the controller has set at time t_k), the following simplification of (2.1.9) is allowed for $n+1$ points in time $t_0..t_n$, at which controller steps are taken ($n \geq 1$):

$$\theta(t) = \theta_0 + \sum_{k=0}^{n-1} \triangle t_k\, \omega_k + (t - t_n)\, \omega_n, \tag{2.1.14}$$

where $t \in [t_n, t_{n+1}]$ and $\triangle t_k = t_{k+1} - t_k$.
For $n = 0$ and $\triangle t = t - t_0$, equation (2.1.9) further reduces to:

$$\theta(t) = \theta_0 + \triangle t\, \omega. \tag{2.1.15}$$

In this case, equation (2.1.10) may also be restated in a simplified form, completing the dynamic model for a robot:

$$\vec{x}(t) = \vec{x}_0 + \begin{cases} v^s \triangle t \begin{bmatrix} \cos(\theta_0) \\ \sin(\theta_0) \end{bmatrix} & \text{, when } \omega = 0 \\[2ex] \frac{v^s}{\omega} \begin{bmatrix} \sin(\omega\triangle t + \theta_0) & - \sin(\theta_0) \\ -\cos(\omega\triangle t + \theta_0) & + \cos(\theta_0) \end{bmatrix} & \text{, otherwise.} \end{cases} \tag{2.1.16}$$

According to equation (2.1.16), the robot is either moving on a straight line ($\omega = 0$) or on the circumference of a circle ($\omega \neq 0$) for any time interval with constant ω and v^s. A trajectory of any robot's path in (simulated) physical space therefore consists of arcs as depicted in Figure 2.3 and Figure 2.4

2.1.4 Properties of the derived Dynamic Models

So far, dynamic models have been derived for all objects to be included in the setting of the simulation.
The difference to many other modelling and simulation-approaches is, that the models derived in the last sections are in *explicit analytical* form. This distinction is of major importance since it allows exact integration of the quantities in question, rather than numerical approximations (as in many other modelling and simulation-approaches) which is usually much more expensive (as far as execution time and accuracy is concerned).
For instance, many simulations rely on numerically integrating the differential equations of the objects motion by methods such as *Runge-Kutta*. Although this is a common and respectable way of dealing with the task it raises many issues, such as choosing the right step-size, stability-issues and the like. Most importantly, numerical integration-methods do not offer a way of conveniently handling events such as collisions.

However, the chosen approach imposes other obligations, namely that of dealing with collisions more cautiously: Since exact integration is possible over arbitrary large time-intervals (as long as no collision or any other event, such as a controller step or the halting of a ball has occurred), the need for precise collision-prediction arises.
This issue will be dealt with in the next chapter.

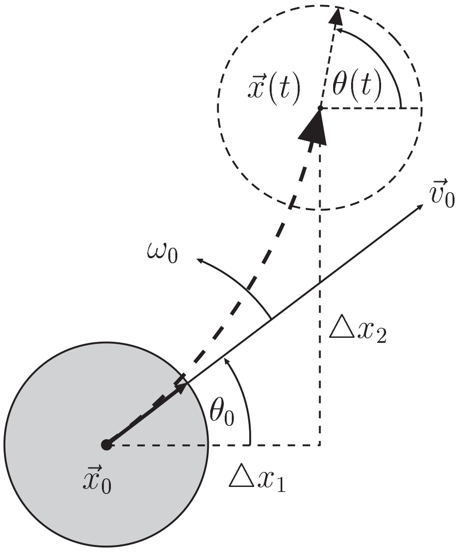


Figure 2.3: Illustration of a piece of the robot's trajectory in space. Its initial position at t_0 is $\vec{x}_0$, θ_0 with tangential velocity $\vec{v}_0$. After a time $\triangle t = t - t_0$ of moving with constant ω and v^s, the robot has traveled $\triangle \vec{x}$ to a new position $\vec{x}(t)$, $\theta(t)$ (along the dashed bold arc).

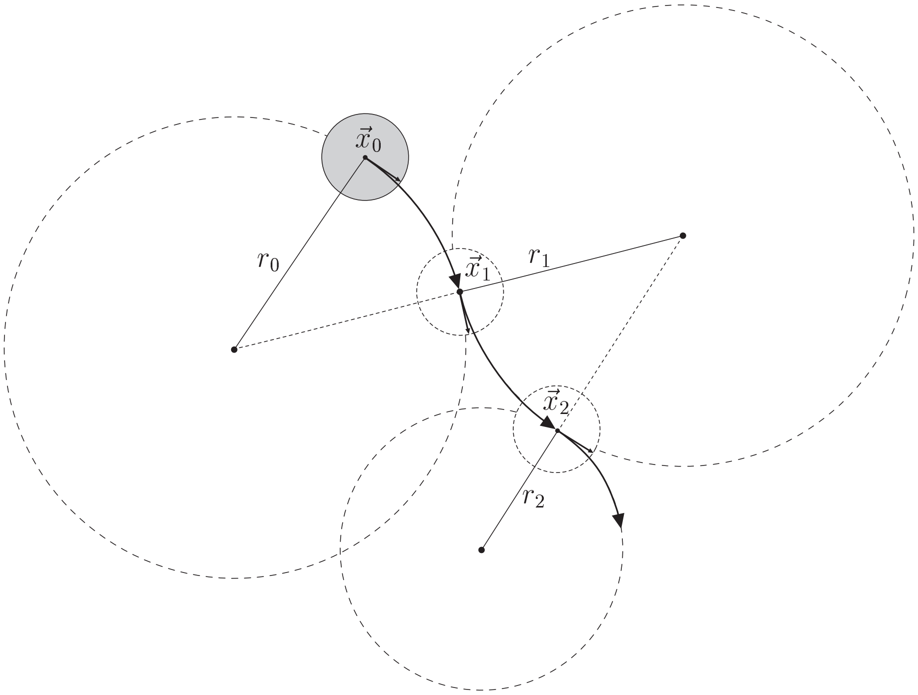


Figure 2.4: The whole trajectory consists of a series of patches shaped just as the arc which is depicted in Figure 2.3. Three snapshots of successive points in time $t_0..t_2$ and corresponding positions $\vec{x}_i$ are shown at which a controller-step has occurred, setting a different ω and v^s each time. The patches are arcs on a circle with radius $r_i = \frac{v^s}{\omega}$.

2.2 The Perceptive Model of a Sensor

In order for a robot-controller to be able to decide upon (any meaningful) actions, it has to be able to perceive the present situation it is situated in. This ability relies on the robot's sensory system. Therefore, the sensory model has to provide the controller with updated input at every controller-step. This is by far the most expensive computation needed in order to simulate any system of interest in the considered type of settings.

Sensor-modeling can be divided into two separate tasks, namely into modeling the geometric properties of a sensor (position in space, field of view, range etc.) as well as modeling the perceptive properties of a sensor. Here it has to be stressed once again, that emphasis is put on developing a reasonable prototypical and abstracted model rather than modeling a real sensory-device with certain (given) properties and capabilities.
Therefore the assumptions underlying this modelling step are being stated explicitly first and then appropriate models are derived from them (as was the case for the dynamic-models).

2.2.1 Geometric Properties of a Sensor

The geometric model of a sensor consists of the following data:

- n_s : number of sensors
- for each sensor $j \in 1..n_s$:
  - (d_j, δ_j) : relative position of sensor j in polar coordinates wrt. the robots position,
  - β_j : main-angle of sensor j's field-of-view,
  - α_j : opening-angle of sensor j's field-of-view.
  - r_j : range of sensor j.

These quantities represent the whole geometric model and are constant since they are defined with respect to the robots position (sensors move in space, but only due to the robots movement - they don't move wrt. the robot's local coordinate system). The geometric model will serve as the basis for developing a model for the perceptive properties of a sensor.
Figure 2.5 and Figure 2.6 illustrate the parameters of the geometric-model and a fully equipped robot with six sensors, respectively.

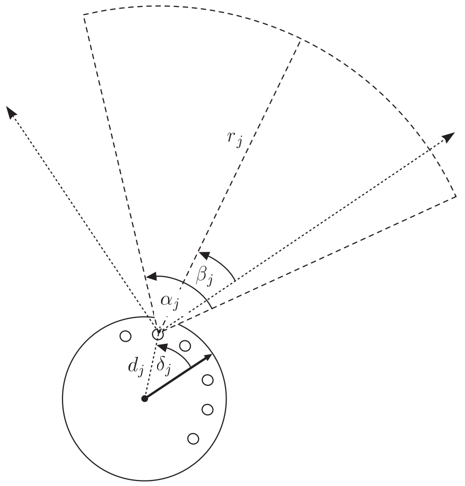


Figure 2.5: Illustration of the geometric model of a sensor. The quantities (r_j, δ_j), α_j, β_j and r_j are defined with respect to the local coordinate system imposed by the robot's position.

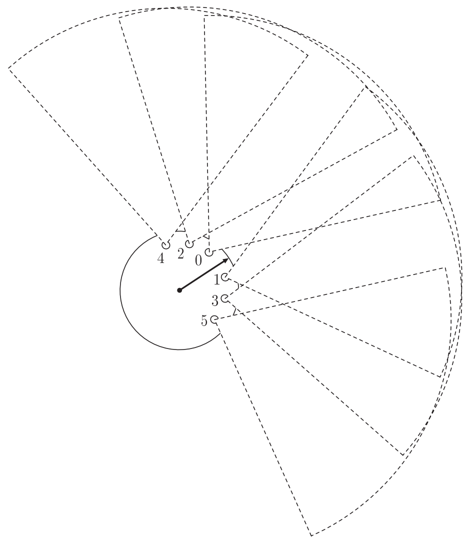


Figure 2.6: Illustration of a typical arrangement of six identical sensors (wrt. their characteristic values α_j, β_j and r_j) with overlapping fields of views.

2.2.2 Claims at and Assumptions about the Model

Before derivation of a perceptive model will be presented, it has to be noted that there are many possible plausible models and by the time of writing, a few have been developed, implemented, tested and rejected - or refined, resulting in the model to be presented here. However, first a prior model will be derived from some starting assumptions to fully appreciate the final model to be presented thereafter.

Underlying Assumptions, Simplifications and Claims

As specified in the model of the sensor's geometric properties, each sensor j is uniquely positioned on a robot, possessing a field of view (specified by α_j,β_j and r_j). Starting from here, the following assumptions are taken as a base for the perceptive model(s) to be derived:

(i) At any point in time t, every sensor should have a disambigous sensor-value $s_j(t)$ connected to it, reflecting the intensity of its perception at that time.

(ii) The sensor-value $s_j(t)$ should be monotonically related to the *distance* of the perceived objects in a decreasing manner, given all other variables remain unchanged (i.e. the sensor-value for an object closing in on the robot may still decrease if the object is moving out of the sensor's field-of-view).

(iii) Given all other variables being equal, the sensor-value $s_j(t)$ should be monotonically related to the *size* of the perceived objects in an increasing manner.

(iv) For any object, moving out of the sensor's field of view (i.e. with increasing deviation of the angle to that object from the sensor's main viewing-angle), the sensor-value $s_j(t)$ should be monotonically decreasing.

(v) The borders of the sensor's field of view should not impose any discontinuities on the sensor value $s_j(t)$ (i.e. an object entering a sensor's field-of-view in a continuous manner should not cause the related sensor-value to be discontinuous at any point in time).

(vi) Finally, the sensor-value $s_j(t)$ should take on values in a predefined interval $[s_{min}, s_{max}]$, where s_{min} and s_{max} are not related to any other sensor's characteristics (particularly not to α_j, β_j and r_j).

2.2.3 A first (naive) Perceptive Model

One of the first approaches to a decent sensory-model has been to refine the sensor's field-of-view with several rays originating at the sensor's position and covering the area in a way which assures, that any object present in the world would be detected by the rays if its intersection with the field-of-view was "significant". Of course, the criterion "significant" lacks precision here, but this model at least assures the possibility (for a given set of objects present in the world) to specify exactly how far the objects can maximally enter the sensor's field-of-view before detection.

With this approach it is also possible to distribute more rays around the sensor's main-angle in order to assure finer resolution in that region, also resulting in higher sensor-values for objects closer to the main-angle (in case rays are weighted equally).
One major problem with the naive model is that discontinuities in sensor-values may be caused by an object, which is just leaving or entering the range of one of the sensor's rays.

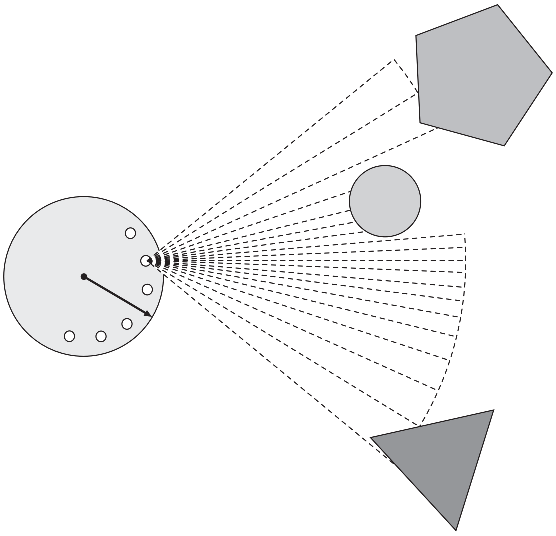


Figure 2.7: Illustration of the naive perceptive-model with static rays. The two polygonal shapes illustrate the possibility of an object within sensor-range not being detected by the sensor, whereas the circle (a ball) is within sensor-range and is also detected by four rays.
The rays are distributed, such that resolution near the main-angle is finer grained than on the borders of the field of view.

2.2.4 A refined Perceptive Model

Since the naive perceptive model suffers from discontinuities caused by a finite numbers of sensor-rays, the model needs further refinement resulting in what is described in the following:

The refined sensor-model works similar to integration of the distance $\rho(\gamma)$ to the closest perceived object at viewing angle γ over the total field of view, much like integration in polar-coordinates, although no functional determinate is needed. The distance $\rho(\gamma)$ hereby serves as an argument to the *distance-modifier function* $D(.)$ and a kernel $K(\alpha, \gamma)$ is multiplied in the integration, resulting in the following equation:

$$s = \int_{-\frac{\alpha}{2}}^{\frac{\alpha}{2}} K(\alpha, \gamma)\ D(\rho(\gamma))\ d\gamma, \tag{2.2.1}$$

where the total field of view is given by the interval $[-\frac{\alpha}{2}, \frac{\alpha}{2}]$.
Appropriate choices of Kernel K and D ensure compliance with items (iv,v) and item (ii), respectively. By choosing K such, that:

$$\int_{-\frac{\alpha}{2}}^{\frac{\alpha}{2}} K(\alpha, \gamma)\, d\gamma \quad = \quad 1 \tag{2.2.2}$$

$$\text{and:} \quad \forall\alpha\,\forall\gamma\ K(\alpha, \gamma) \quad \geq \quad 0 \tag{2.2.3}$$

and choosing $D(x)$ such that it is monotonically decreasing with

$$\forall x \; 1 \geq D(x) \geq 0, \tag{2.2.4}$$

compliance with item (vi) is also ensured. In particular the sensor-values are always normalized, such that $s_j(t) \in [0, 1]$, in this case.
Specifically, the following functions are chosen from here on:

$$K(\alpha, \gamma) = \begin{cases} \frac{1}{\alpha} \cos\left(\frac{2\pi\gamma}{\alpha}\right) + 1 & \text{, when } \gamma \in [-\alpha, \alpha] \\ 0 & \text{, otherwise} \end{cases} \tag{2.2.5}$$

$$D(x) = e^{-\lambda \tan\left(\frac{\pi \, x}{2 \, x_{max}}\right)}, \tag{2.2.6}$$

where $\lambda \geq 0$ is a parameter determining how distance is weighted by the *distance-modifier function* (as illustrated in Figure 2.9).
Typical shapes for $K(.,.)$ are depicted in Figure 2.8.

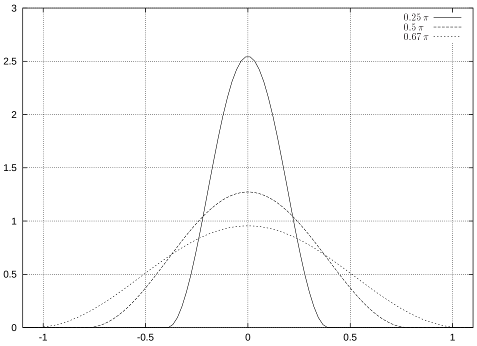


Figure 2.8: Three different kernels corresponding to three different opening-angles α of a sensor.

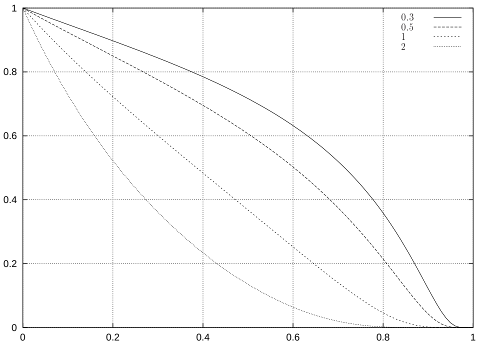


Figure 2.9: Illustration of the *distance-modifier-function* for different values of λ.

2.2.5 Implementation of the refined Perceptive Model

Of course actual evaluation of the refined sensor-model cannot be done analytically for arbitrary shapes. For the simulator to be developed a numerical method is utilized, namely *Gauß*'s method of integration (of fifth order), which is found in many books on numerics as for example in [Sto93]. This special method of numerical integration usually provides very exact results (as compared with other common techniques) and its precondition of providing special (not equally-spaced) points of support is not a drawback, here, but is suited very well for the problem at hand since the individual rays can be positioned arbitrarily.

In fact, the ray's orientation is dynamic and has to be recomputed at every controller-step in any way, such that integration is further refined by applying *Gauß*'s method separately to every maximum subinterval of the total field of view which is occupied by exactly one object and just setting all other subinterval (with no object in view) to 0.
Thus, the key idea is to still use a finite number of rays to approximate the sensor-value, but which will be allocated and positioned dynamically depending on the objects within sensor-range. Hereby, the interval $[-\frac{\alpha}{2}, \frac{\alpha}{2}]$ representing the sensor's field of vision is being partitioned into subintervals containing either one or no perceived object.

2.2.6 Comparison and Evaluation of the Models

The procedure described in the last subsection is of rather technical nature and will not be presented in detail, rather three examples (a),(b) and (c) are given, where the refined sensor-model is compared to the naive.
In Example (a) a robot slowly approaches a ball frontally, in (b) the robot turns closely in front of a (passive) ball and in (c) the ball is placed at different positions, covering the whole field of view of one sensor while the robot remains stationary. In all examples, values for only one sensor are recorded. The experiments (as well as their outcome) are illustrated in Figure 2.10 through Figure 2.16 and clearly show the refined sensor-model to be superior, satisfying all claims made at the beginning of this section.

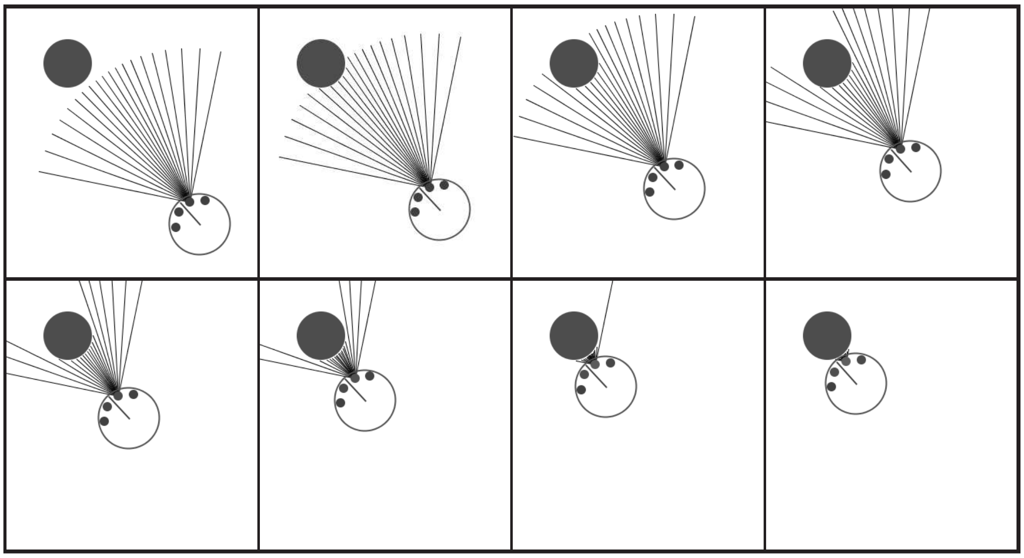


Figure 2.10: Example (a) is illustrated for the naive sensor-model with predefined and statically distributed rays (shown for only one sensor): The sequence depicts the robot while frontally approaching a ball.

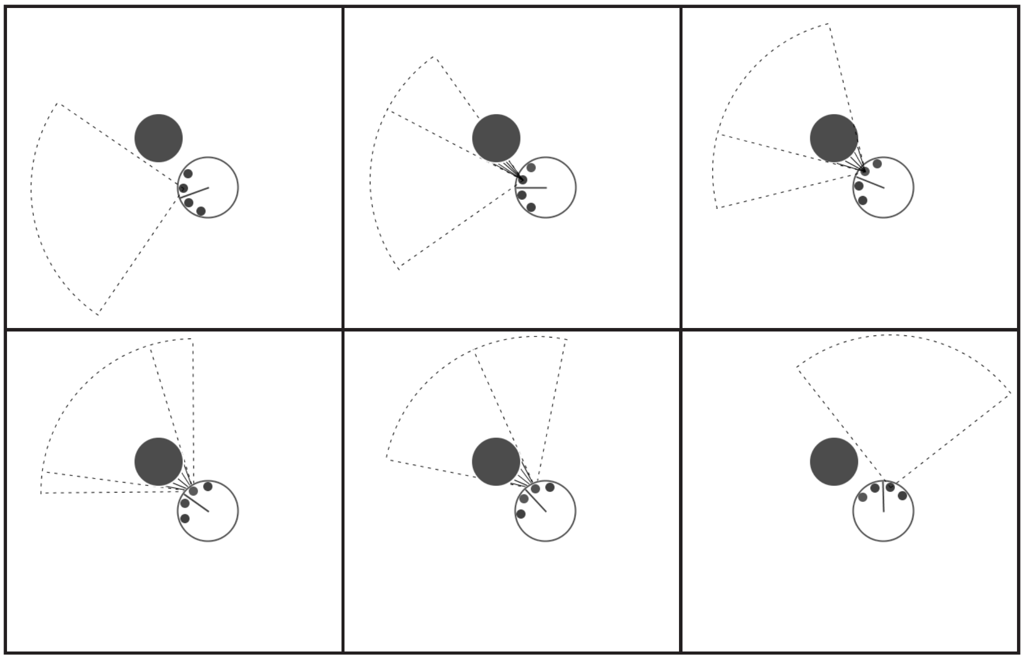


Figure 2.11: In example (b), a robot is slowly turning in front of a passive ball. This time a sequence for the refined-sensor model is depicted, where the partitioning of the field-of-view and dynamic placement of the rays is shown.

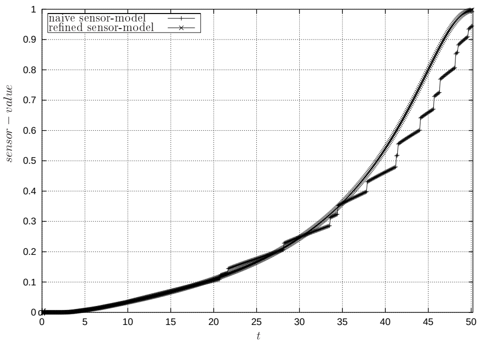


Figure 2.12: Example (a): With respect to the claims at a sensor-model, the refined model proves to be clearly superior over its naive counterpart.

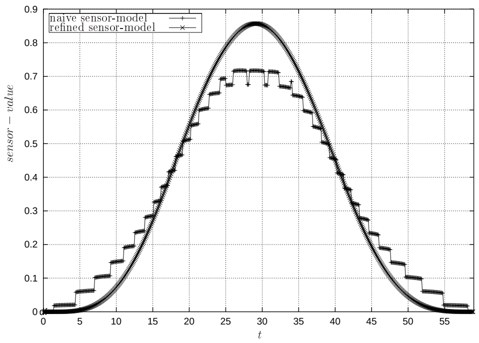


Figure 2.13: Example (b) illustrates the superiority of the refined sensor-model for a robot turning in front of a ball.

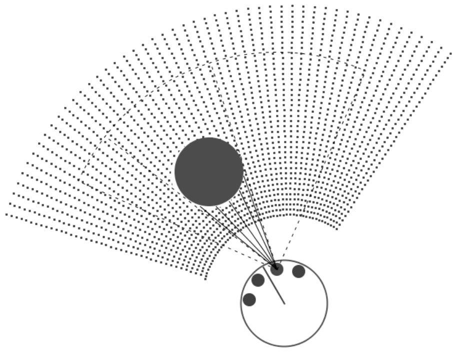


Figure 2.14: In Example (c), a ball is placed at different positions within a sensor's field-of-view. Only every fourth position is shown.

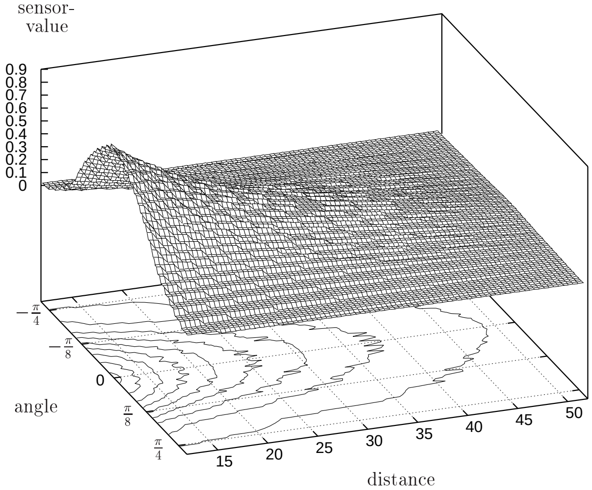


Figure 2.15: Illustration of the outcome of Experiment (c) for the naive perceptive-model. The discontinuities caused by the individual rays are clearly visible. A contour plot (with linear interpolation) representing various levels of sensor-values is projected onto the x-y plane.

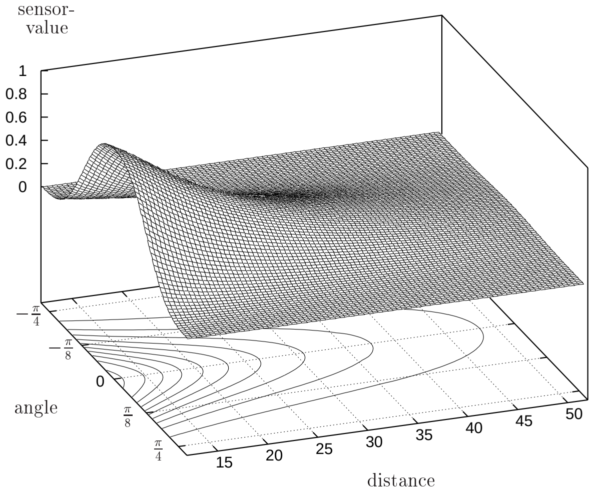


Figure 2.16: Outcome of Experiment (c) for the refined perceptive-model. The surface over the sensor's whole field-of-view appears quite smooth and the contours of equal sensor-values are shaped ideally.

Chapter 3

Modeling Interactions among Objects

As the last chapter was concerned with derivation of physical models for the single objects (components present in the world or *composed system*) this chapter will deal with modeling the *composed system* by considering the interactions among components (Strictly speaking, one foundation for interaction has already been given in the last chapter, when the sensory model was derived).

3.1 The Guard

So far the proposed *component models* don't suffice to build a *composed-model*, since it is desired to capture not only a component's behavior as a stand-alone entity, but also to model the interaction among multiple components. One further entity (a *guard*) will be introduced therefore, which will not correspond to a physical object (as a robot or a ball), but which will fit the post of caring for the physical object's collisions and sensor-perceptions. Only one such component is to be included in any world, dealing with all possible interactions of objects in that world.

More specifically, the guard's job is to care for collisions between objects as well as to determine the sensor-values at any point in time desired.
The first task, caring for collisions, can be decomposed into two subtasks:

- computing the time of the next collision taking place within the world (and of course, determining the involved objects as well as their positions on that collision-event)
- determining the way in which the involved objects state is effected by that collision.

3.2 Collision Prediction

The collision-prediction subtask for two dynamic objects a and b can equivalently be reformulated in mathematical terms as solving the equation:

$$\|\vec{x}_a(t_{coll}) - \vec{x}_b(t_{coll})\| = r_a + r_b, \tag{3.2.1}$$

with constraint:

$$\langle \vec{n}, \vec{v}_a(t_{coll}) \rangle - \langle \vec{n}, \vec{v}_b(t_{coll}) \rangle > 0, \tag{3.2.2}$$

where $\vec{n}$ denotes the *collision-normal* as depicted in Figure 3.1 and $\vec{v}_a$,$\vec{v}_b$ denote the tangential velocities of objects a and b, respectively.
Given the case that a collision will take place at all, solving the equation (for t_{coll}) will yield two distinct solutions in general. Of these, the minimum positive solution is the desired prediction.

However, it has to be stated here that only in the case of ball-ball collisions a solution has been found in analytical form. For other combinations (i.e. ball-robot and robot-robot collisions), the equation's solution is approximated by numerical methods which have been elaborated using quite a few of the algorithms from the rich collection of articles on computational geometry, found in [Gla90] and [Arv91].
The developed algorithms perform quite well in practice and can be witnessed in action when the conducted experiments are presented in chapter 5 and 6.

Collision prediction in the case of a dynamic object colliding with a static object is a little easier (since only one is actually moving) and analytical solutions have been found in all cases by predicting collisions of the dynamic object with all corners as well as all edges of the static object.
Design of the algorithms will not be discussed here in any more detail and neither will the derivation of analytical solutions, since it is considered a rather technical issue which does not contribute much to the the modeling aspects in focus.

3.3 Collision Handling

Collision handling has been identified as another responsibility of the guard, it refers to the task of determining the effect of a predicted collision and its realization.
First, only collisions with two involved objects are considered in this section, more objects concurrently colliding with one another will be dealt with in subsection 3.3.6 on page 30.

3.3.1 Collisions with two Dynamic Objects involved

The collision-handling implemented applies the (perfectly) elastic collision model (e.g. see [HKl89] and [Woa00]) which views collisions as taking place at one point in time and is concerned with the issue of determining how the experienced impact is reflected in the change of velocities immediately after the collision, obeying the principles *conservation of kinetic energy* and *conservation of impulse.*

In more concrete terms, when two objects a and b with mass m_a,m_b and (tangential) velocities $\vec{v}_a$,$\vec{v}_b$ collide (as depicted in Figure 3.2), their experienced impulsive forces result in new velocities $\vec{v}_a^{coll}$ and $\vec{v}_b^{coll}$ which are given by (only for object a):

$$\vec{v}_a^{coll} = \vec{v}_a + \frac{2m_b}{m_a + m_b} \langle \vec{n}, \vec{v}_b - \vec{v}_a \rangle \vec{n}, \tag{3.3.1}$$

where $\vec{n}$ denotes the *collision normal*, which is given by the line connecting the centers of the two colliding objects (as illustrated in Figure 3.1).

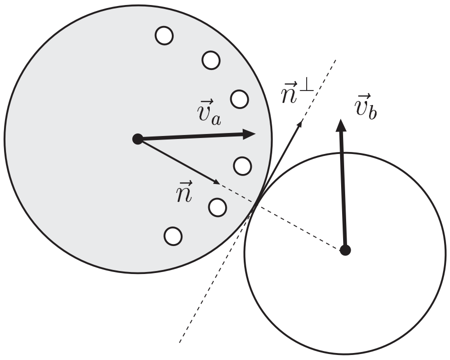


Figure 3.1: The *collision-normal* $\vec{n}$ is given by the line connecting the centers of the two colliding objects. It's orthogonal counterpart is denoted by $\vec{n}^{\perp}$.

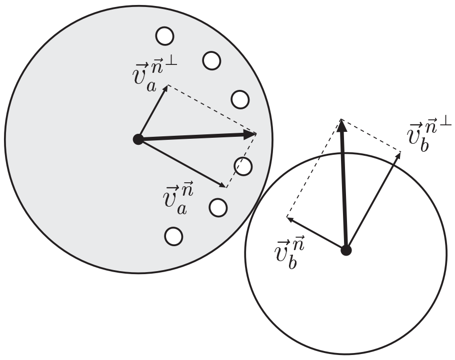


Figure 3.2: The (tangential) velocity of an object can be projected onto $\vec{n}$ and $\vec{n}^{\perp}$. The projections are given by $\vec{v}^{\vec{n}}$ and $\vec{v}^{\vec{n}^{\perp}}$, respectively, where $\vec{v}^{\vec{n}} = \langle \vec{v}, \vec{n} \rangle \vec{n}$.

3.3.2 Collisions with a Static Object involved

Clearly the discussion above is useless if a static object is involved in a collision, since it is just defined *static* and will thus not move no matter what force is applied to it.

However, the static object's mass may be considered in the limit: Say b is the static object then, as $m_b \to \infty$, equation (3.3.1) just yields:

$$\vec{v}_a^{coll} = \vec{v}_a - 2 \langle \vec{n}, \vec{v}_a \rangle \vec{n}, \tag{3.3.2}$$

where $\vec{n}$ is now the *surface-normal* of the static object in case the dynamic object is colliding with an edge or, in case it is colliding with a corner, lies on the line connecting the object's center with that corner.
In case of an edge-collision, equation (3.3.1) just states the well known law: *Angle of incidence equals angle of reflection.*

3.3.3 Effect of a Collision on a Ball

In the case when one collidee is a ball, the collision-effect is readily given by the above equations. It's velocity just has to be changed accordingly and new collision predictions are computed with the updated values.

3.3.4 Effect of a Collision on a Robot

In the case when one collidee is a robot, the world is not just as simple. There are more relevant forces acting on a robot than just the experienced impulsive force due to the collision, namely the force exerted by its own drive and an angular collision-force (which was just neglected in the case of a ball since its orientation is meaningless and angular velocity is not considered in the dynamic model of a ball).

Here again, it is assumed of the robots motor(s) to be sufficiently powerful of providing constant wheel-rotation according to the values v^s and ω, set by the controller (meaning, an ideal drive is assumed).
However, due to the exerted impulsive force it is assumed that the robot's wheels will slip for a (possibly very) short period of time. During this time-interval $(t_{coll}, t_{stopslip}^{ideal}]$ the robot initially slips according to $\vec{v}_{coll}$, as discussed above.

An Ideal Model for the Slipping-Path of a Robot

Since the robot's wheels keep turning at constant speeds during the *slipping-interval* (according to the assumptions made above), its experienced force of friction due to slipping is approximately proportional to the difference in driving-velocity and slipping-velocity: $F = \mu_f \, m \left\| \vec{v}_{drv} - \vec{v}_{slip}^{ideal}(t) \right\|$, where any effects due to a change in orientation have been neglected (meaning $\vec{v}_{drv}$ is approximated as the tangential velocity at time t_{coll}).

It is thus concluded by inspection that the slipping-velocity $\vec{v}_{slip}$ will approach $\vec{v}_{drv}$ exponentially. (Which may be viewed as a generalization of the dynamic ball-model, where the driving force is just $\vec{0}$.[1])

Of course a robot will not keep slipping forever, but will at some point catch grip again and continue to drive. (When the robot's momentum has dropped sufficiently low and $\vec{v}_{slip}(t)$ has approached $\vec{v}_{drv} = \vec{v}(t_{coll})$ sufficiently close.) The following equations express the described ideal model in mathematical terms, where $\triangle v_{stopslip}$ represents the difference in velocities when the robot's wheels catch grip again, determining the time of slipping $\triangle t^{ideal}_{slip} = t^{ideal}_{stopslip} - t_{coll}$:

$$\triangle t^{ideal}_{slip} = \frac{1}{\mu_f} \ln\left(\frac{\|\vec{v}_{drv} - \vec{v}_{coll}\|}{\triangle v_{stopslip}}\right) \tag{3.3.3}$$

$$\vec{v}^{ideal}_{slip}(t) = \vec{v}_{drv} - e^{-\mu_f \triangle t}(\vec{v}_{drv} - \vec{v}_{coll}), \tag{3.3.4}$$

where $\triangle t = t - t_{coll}$.
The ideal slipping-model is illustrated by an example (in one dimension) in Figure 3.3.

Proposed Model for the Slipping-Path of a Robot

Although quite a few (possibly) simplifying assumptions have been made so far, complete attribution of all the above statements would still result in too complex of a model to be dealt with, here.
Therefore a (further) simplified model is derived from the *ideal* model which allows for exact collision prediction within the framework presented so far. Concretely, the collision-prediction algorithms briefly mentioned in section 3.2 on page 19 will suffice for the model proposed here. The ideal model still serves as a base for derivation of the proposed model:

Whenever a collision occurs, a robot's state is immediately switched to slipping and its velocity is set to $\vec{v}_{coll}$ as was derived earlier.
This velocity is held constant ($\vec{v}_{slip}(t) = \vec{v}_{coll}$) over the whole slipping-interval resulting in an overestimation of slipping velocity for times $t \in (t_{coll}, t_{stopslip}]$. In turn, the total time of slipping $\triangle t_{slip}$ is modified such that, neglecting angular velocity, the total change in position at the end of the slipping-

[1] Although the argument is quite similar to how the dynamic model for a ball was derived, the coefficient of friction μ_f for a robot is assumed to be much higher than for a ball. Afterall, it represents the characteristic resistance of a *slipping* robot as opposed to a *rolling* ball.

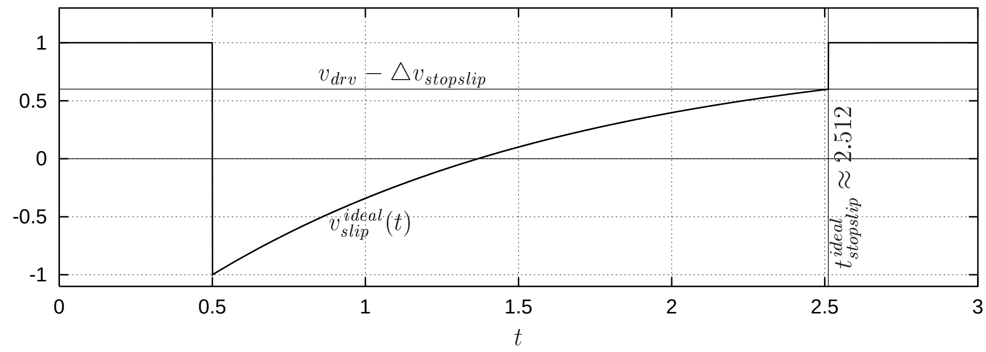


Figure 3.3: Illustration of the ideal model of a slipping robot in one dimension: At time $t_{coll} = 0.5$, the robot (frontally) collides with a static object: The exerted impulsive force of the collision causes the robot to start slipping with initial slipping velocity $v_{coll} = v_{slip}(t_{coll}) = -v_{drv} = -1$. It does so until its slipping-velocity $v_{slip}(t)$ has approached v_{drv} sufficiently close after $\triangle t_{stopslip} \approx 2.012$ time-units. The constants are: $\mu_f = 0.8$, $\triangle v = 0.4$ and thus, at time $t_{stopslip}^{ideal} \approx 2.512$ the robot's wheels catch a grip again, causing the robot to continue driving. The total length of the slipping path is 2 units.

interval is just the same as for the ideal model or more precisely:

$$\int\limits_{t_{coll}}^{t_{stopslip}} \|\vec{v}_{drv} - \vec{v}_{slip}(\tau)\| \; d\tau \;=\; \int\limits_{t_{coll}}^{t_{stopslip}^{ideal}} \left\|\vec{v}_{drv} - \vec{v}_{slip}^{ideal}(\tau)\right\| \; d\tau. \tag{3.3.5}$$

By substituting equation (3.3.4), the righthand-side simplifies to:

$$\int\limits_{t_{coll}}^{t_{stopslip}^{ideal}} \left\|\vec{v}_{drv} - \vec{v}_{slip}^{ideal}(\tau)\right\| \; d\tau \;=\; \frac{\|\vec{v}_{drv} - \vec{v}_{coll}\|}{\mu_f} \left(1 - e^{-\mu \triangle t_{slip}^{ideal}}\right). \tag{3.3.6}$$

Since the left-hand side of equation (3.3.5) just equals $\triangle t_{slip} \, \|\vec{v}_{drv} - \vec{v}_{coll}\|$, dividing the righthand-side by $\|\vec{v}_{drv} - \vec{v}_{coll}\|$ and substituting equation 3.3.3 yields the total slipping-time $\triangle t_{slip}$ of the proposed model:

$$\triangle t_{slip} = \frac{1}{\mu_f} \left(1 - \frac{\triangle v_{stopslip}}{\|\vec{v}_{drv} - \vec{v}_{coll}\|}\right) \tag{3.3.7}$$

In case $\|\vec{v}_{drv} - \vec{v}_{slip}\| < \triangle v_{stopslip}$, a minimum (positive) slipping-period $\triangle t_{slip}^{min}$ is always enforced.
The proposed dynamic model for a slipping robot is quite simple:

$$\vec{x}(t) = \vec{x}(t_{coll}) + \triangle t \, \vec{v}_{slip} \;, \tag{3.3.8}$$

for any $t \in [t_{coll}, t_{stopslip}]$ with $\triangle t = t - t_{coll}$.
The proposed model is illustrated in one dimension by way of example in Figure 3.4.

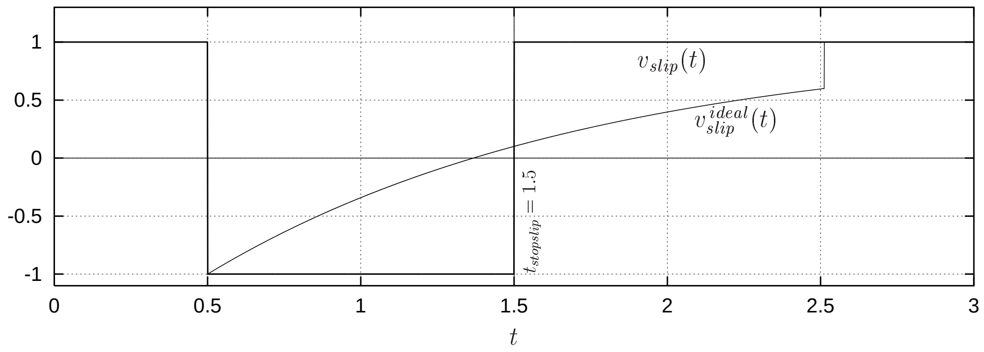


Figure 3.4: Illustration of the proposed model of a slipping robot in one dimension: At time $t_{coll} = 0.5$, the robot (frontally) collides with a static object - the same setting as in Figure 3.3 is considered. The total slipping interval evaluates to $\triangle t_{slip} = 1$, such that the induced change in position after the slipping-interval (with $v_{slip} = -1$) is the same as for the ideal model. The ideal model $v_{slip}^{ideal}(t)$ is included in the plot for comparison.

Angular Effects of a Collision

A colliding object in the real world generally experiences forces which cause changes in translation as well as orientation. So far angular effects have not been considered, in fact they have been neglected altogether in the case of a colliding ball and also as far as they effect the path of a slipping robot. Although a robot's slipping path is simplified to be a straight line-segment, angular collision-forces acting on a robot *are* considered as far as they effect the robot's orientation *after* slipping.
This change in orientation $\triangle\theta_{slip}$ over the whole slipping-interval $\triangle t_{slip}$ is just set to:

$$\triangle\theta_{slip} = \triangle t_{slip}\,\omega_{slip}, \tag{3.3.9}$$

where ω_{slip} is modeled constant over the slipping-interval (similar to $\vec{v}_{slip}$). In case of two dynamic objects it is given by:

$$\omega_{slip} = \frac{2m_b}{r_a(m_a + m_b)} \left\langle \vec{n}^{\perp}, \vec{v}_b - \vec{v}_a \right\rangle \vec{n}^{\perp}. \tag{3.3.10}$$

In case a robot is colliding with a static object, the corresponding equation is:

$$\omega_{slip} = -\frac{2}{r_a} \langle \vec{n}^{\perp}, \vec{v}_a \rangle \, \vec{n}^{\perp}. \tag{3.3.11}$$

Angular velocity produced by the robot's drive is not accounted for in the model for slipping-paths.
When multiple collisions are occurring in short time-intervals (such that the robot is still slipping while another collision occurs), the angular velocities of previous slipping intervals are applied before a new slipping-interval is enforced.

Illustration of the Model for a Slipping Robot

Only partial derivation (and thus justification) has been given for the proposed model of a slipping robot, to at least provide an intuitive justification it is compared with the ideal model by way of example, given in Section 3.3.4.
Since the proposed model takes angular velocity into account as far as it effects the robot's orientation at the end of the slipping-interval, the ideal model has been extended to do also for the examples.
Hereby, angular velocity while slipping is initially set to ω_{slip} as in the proposed model and then assumed to exponentially approach ω (set by the controller) just as was the case for the tangential slipping-velocity. The change in orientation caused by angular velocity is added at the end of the slipping interval for the sake of comparability of the two models.

In Figure 3.5 on page 26, three prototypical settings are considered to illustrate the difference between the ideal and the proposed model of a slipping-path.
As expected, the ideal model provides smoother trajectories of a slipping robot. Also noteworthy is the fact that for times $t > t^{ideal}_{stopslip}$, the proposed model is just a shifted version of the ideal model, since after $\triangle t^{ideal}_{slip}$ time-units, the change in orientation caused by both models is equal. This is the case because of the choice made for $\triangle t_{slip}$ in equation (3.3.7).
In Subfigure (b), equation (3.3.5) can also be seen in effect: The thin trajectories correspond to the ideal and proposed model without considering angular-velocity caused by the collision. Since additionally the robot's drive is set to $\omega = 0$, the trajectories are identical for times $t > t^{ideal}_{stopslip}$.

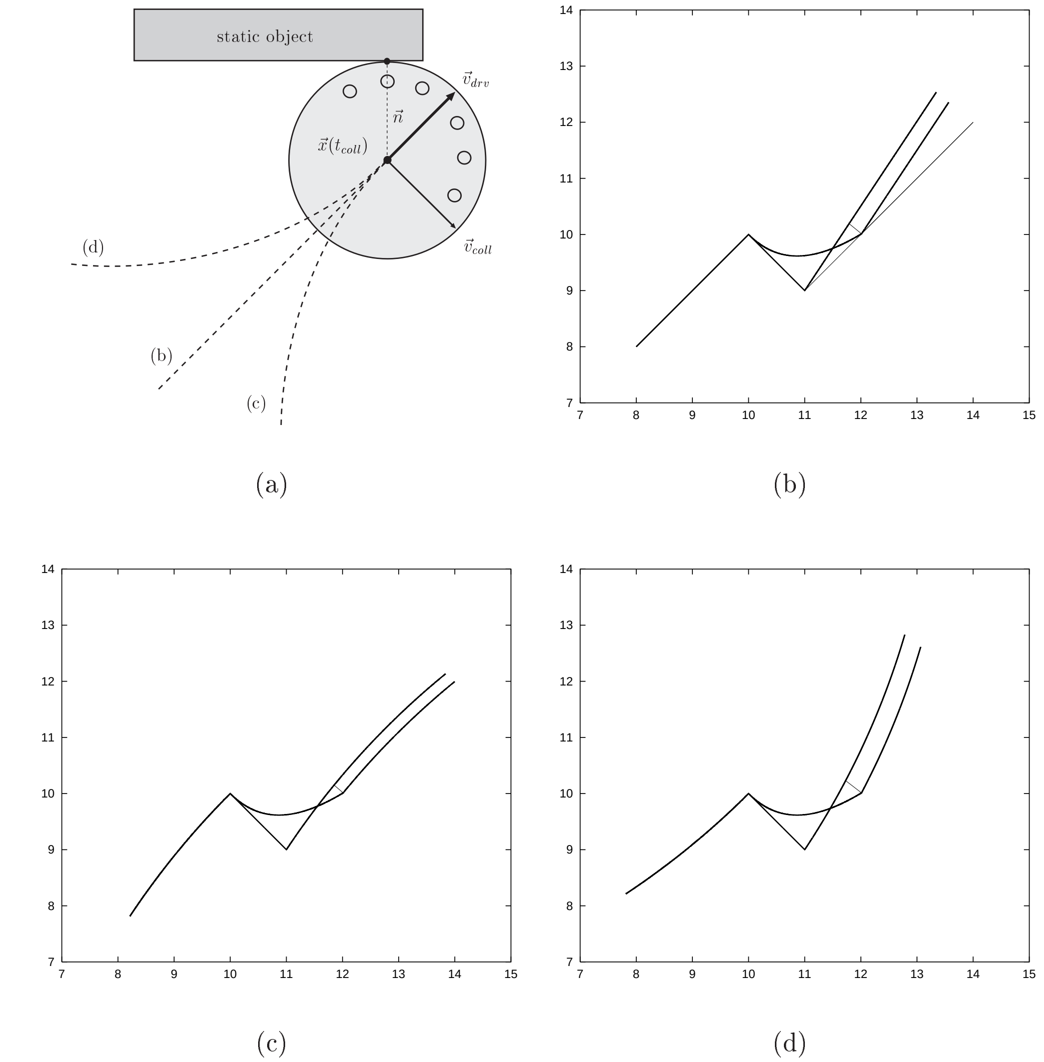


Figure 3.5: Subfigure (a) illustrates three different settings of a colliding robot corresponding to Subfigures (b),(c) and (d). In all three settings, the robot collides with a static object at the same position and orientation driving at the same speed v^s. The only difference is in angular velocity ω.
Subfigures (b) through (d) depict a piece of the robot's trajectory including the slipping-interval, the corresponding angular velocities (set by the controller) are $\omega_b = 0$, $\omega_c < 0$ and $\omega_d > 0$.
The slipping-path forming the straight line-segments correspond to the proposed model, whilst the smooth trajectories (while slipping) correspond to the ideal model.
For the case $\omega_b = 0$, depicted in Subfigure (b), the trajectories without considering angular-slipping-velocity have been plotted as well (thin lines).
The thin straight line-segments, shown in all three subfigures, connect the trajectories of the ideal and proposed model at time $t^{ideal}_{stopslip}$ and illustrate the difference in position between the two models for times $t > t^{ideal}_{stopslip}$.

3.3.5 Summary of Collision Handling and Examples

Collision handling in the case of a ball (i.e. determining the ball's response) has turned out simple, whereas determining the response of a colliding robot proved to be more tricky: The slipping-interval, given by $(t_{coll}, t_{stopslip}]$ is to be interpreted as the time between a collision (causing the robot to slip) and the point in time, when the robot's wheels catch a grip again (and the robot continues to drive). After completion of the slipping-interval, the robot's state is immediately switched from *slipping* to *driving* again.
Clearly, the model of collision handling (in the robot-case) is not as elaborated as the dynamic models derived earlier. Comfort may however be taken in the assurance that slipping-intervals merely represent collision responses which only occur during short time-intervals.

In return for the model's simplicity, collision prediction as discussed in the beginning of this chapter may still be performed in an exact manner within the existing framework, meaning no further cases of collision-types have to be considered than those which have already been mentioned (which would not be the case if the ideal model was considered for simulation). Thus, multiple collisions within short time-intervals or almost constant contact between components can be handled easily.
Three prototypical experiments will be presented next, which, as is argued here, give at least intuitive evidence that the essentials of real-world physics have been approximated reasonably well.

The first experiment, shown in Figure 3.6 is conducted for a robot moving on a slightly curved trajectory which is occupied by a static object. After first contact the robot effectively *slides* along an edge of the object until it has reached its corner and continues to drive unhindered. Hereby, the *angular* collision-force's effect is illustrated quite well: Though the robot's angular (driving) velocity ω is positive (and held constant for the experiment), it's orientation after sliding along the edge is smaller than when it first hit the wall.

The second experiment (Figure 3.7) shows two robots colliding with one another. Since they are almost frontally colliding in the beginning, quite a few collisions have to be predicted and handled until they separate (for the time being) on independent paths again.
It can also be observed that, since robot b's velocity v_b^s is only half of a's velocity, it is pushed back for a while, whereas for robot a it is illustrated how its orientation is changed by a large amount at the time of separation due to angular collision-force (considering that $\omega_a = 0$).

In the third experiment it can be observed in Figure 3.8 how a robot effectively *pins* a ball in front of a static object until the ball is released with high velocity.
It is noted here, that in case the robot were exactly frontally colliding with the ball (on a trajectory perpendicular to the static-object's edge), the dynamics and collision-models given so far would determine everlasting collisions resulting in unrealistically high ball-velocities.
In reality however (which is to say in the actual implementation of the models on a computer), even small numerical errors take care of this problem by providing some initial inexactness which is propagated and amplified by successive computations, causing the ball to be released sooner or later.
It is noteworthy to mention that this is the only case encountered where (omnipresent) numerical errors are actually useful, whereas in most other occasions great care has to be taken in order to avoid harmful effects of numerical errors of various kinds.

It is argued that the experiments which have been presented give meaningful evidence in favor of the conjecture that the essentials of real-world physics have been captured by the proposed models.
Deficiencies are mainly due to the slipping-path-model's simplicity, which causes the trajectories to look somewhat jagged.

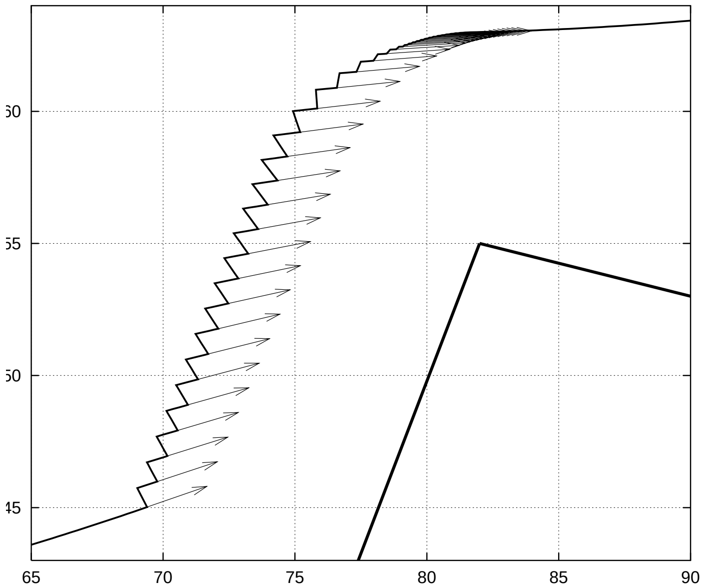


Figure 3.6: Illustration of a robot *sliding* along the edge of a static object. The robot's radius is $r = 8$, it's angular velocity is positive.

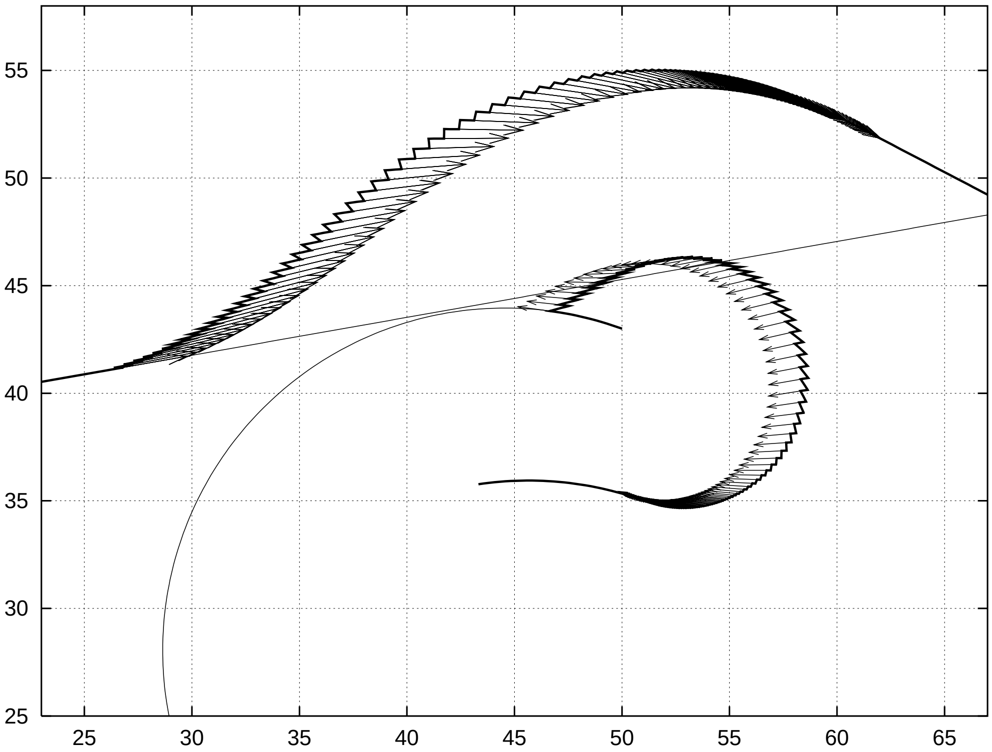


Figure 3.7: The trajectories of two colliding robots are shown (thick lines) as well as their tangential velocities at collision-times (arrows) and their trajectories in case no collision(s) had occurred (thin lines). Robot a (top trajectory) and b are driving according to controller-values $\omega_a = 0$, $\omega_b > 0$ and $v_a^s = 1$, $v_b^s = 0.5$. Their mass is equal and so is their radius: $r_a = r_b = 10$.

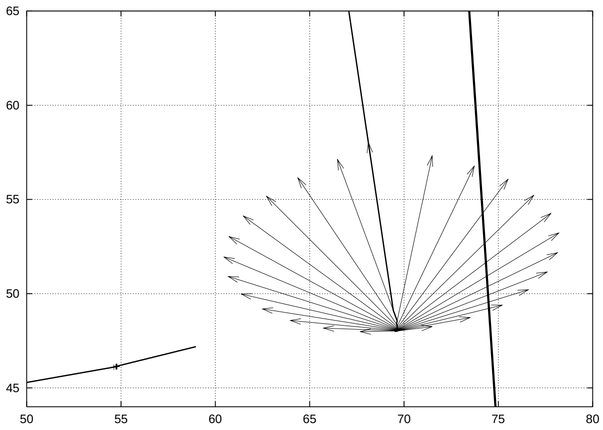


Figure 3.8: Illustration of a robot colliding with a passive ball in front of a static object. Since the ball's mass is about 2 orders of magnitude smaller than the robot's, it can be observed that it is only shortly held back in its trajectory, whereas the ball's final velocity is changed by a much higher amount than the robot's.

3.3.6 Collision Handling for more than two Objects

While the first part of this section was concerned with modelling simple collisions (that is to say collisions between various combinations of only two colliding objects), the more general case of more objects concurrently colliding with one another has to be considered also.
This case can be broken down to collision handling for two objects. The following procedure takes care of this case:

step0: An ordered list of collision predictions (T_{coll}) is initialized for all combinations of objects i, j (disregarding combinations of two *static* objects), where collision-prediction is performed as discussed and the list $T_{coll} = \left[t_{coll}^{i,j}\right]$ is ordered according to the values of t_{coll}

step1: Whenever the simulation-time equals a predicted collision-time for two objects, a set of collision-pairs P_{coll} is initialized with *all pairs* of objects colliding at that time.

step2: For all pairs in P_{coll}, a simple collision handling-step is taken (in arbitrary order) as discussed. The corresponding pair is removed from the list.

step3: When there are no more pairs in P_{coll}, a new collision-prediction step is taken and T_{coll} is updated, where predictions of present collisions are also considered.
The procedure continues with **step1**.

Despite arbitrary complex situations which might arise in the turn of a simulation, collision handling can always be broken down by the procedure above. An illustration of this fact is given in Figure 3.9, showing six colliding balls. This example also perfectly illustrates the principle *conservation of impulse*.

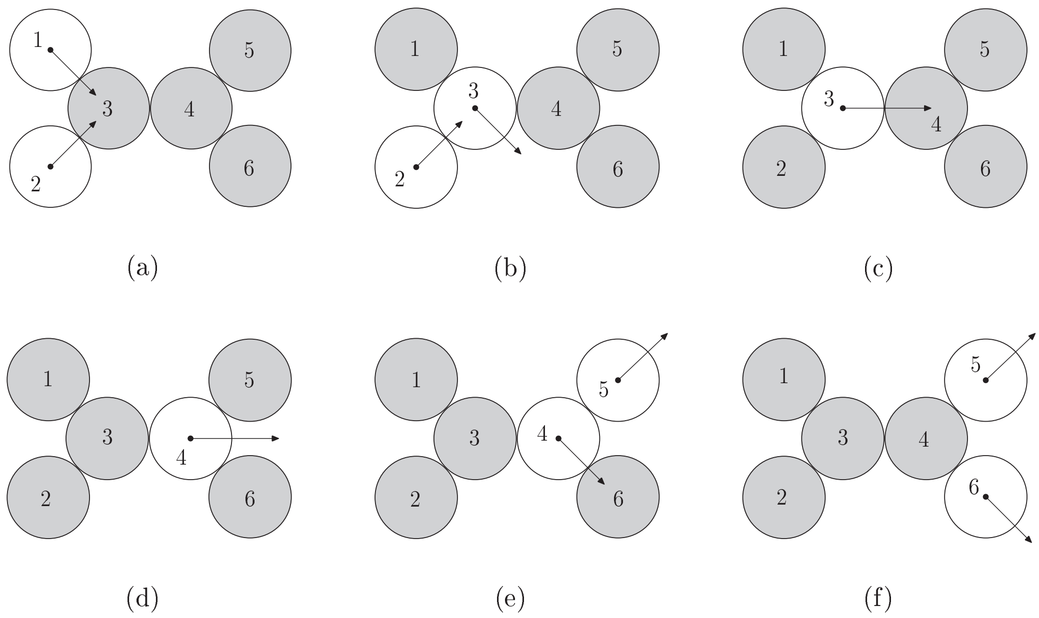


Figure 3.9: Subfigures (a) through (f) illustrate a sequence of collision-handling in case of multiple simultaneous collisions. In each subfigure balls with an impulse different from zero are depicted with a white interior, whereas balls with an impulse of zero are shaded, all balls have the same mass. Velocity is shown by a vector originating at the center of the corresponding object.
In Subfigure (a), balls 1 and 2 have concurrently hit ball 3. The procedure given for general collision handling states that collisions (1,3) and (2,3) are to be handled in arbitrary order (the subfigures depict a sequence ordered according to the ball's indices). After both collisions are handled, shown in Subfigure (c), a new collision prediction step is taken, where collision (3,4) is detected. After handling of collision (3,4) the whole procedure continues likewise until, finally, balls 5 and 6 are given an impulse in Subfigure (f).
Altogether two more collision-prediction-steps have to be performed after collisions (1,3) and (2,3) have been handled, although simulation-time has not advanced at all.

Chapter 4

Implementation of the Simulator

Up to this point a complete collection of mathematical models for all entities in the world and their interaction has been developed.
This chapter is now concerned with integrating these models into one coherent framework which will be referred to as the *composed system* or *composed model*, of the *component models* (balls, robots, static objects and guard). The framework will be imbedded in an abstract formalism, which is nevertheless very well suited for actual execution on a computer.
Finally an object-oriented, pseudocode-style, overview of the implementation is given, which directly applies the above mentioned formalism.

4.1 Discretization of the Proposed Models

The *component models* presented so far are of continuous nature. This may be an adequate property for mathematical models, but will clearly not allow immediate simulation on a computer, since continuous implementations are not possible on digital computers.
For dynamical systems (of which the *composed system* in focus is an instance of), discretization is generally done (at least) for the time-domain.[1]
Two major types of *system-specifications* exist, which allow for (computer-executable) discretization (and in general approximations) of a continuous system.
All existing implementations and formal specifications of discretized continuous systems can be classified as belonging to one of these types:

[1] Of course, any simulation using computer-representation of continuous variables introduces additional discretization (in those variables domains) due to limited storage of variable-representations and computers in general. However this issue is suppressed here.

(i) *Discrete-Time System Specification (DTSS)*: This approach just discretizes time according to a given rule (often, constant time steps are taken). It may be termed the naive approach, which is instantiated by numerous applications including many numerical integration methods with static step-size. It is easy to implement and therefore often chosen for various purposes.
Most discretizations which do not specifically address the topic of *choosing* a discretization-method fall under this category.

(ii) *Discrete-Event System Specification (DEVS)*: This approach discretizes time dynamically according to predicted *events* for the system in question. Simulations which adopt this approach are often also called *event-driven-simulations*.
In some sense, this approach is a generalization of the DTSS-approach (DEVS degenerates to DTSS if the definition of *events* and their prediction just yield a fixed discretization-rule).
Depending upon the fact, whether or not the notion of *events* is applicable to the system in question, DEVS often offers superior characteristics (in terms of performance vs. accuracy), but is usually harder to handle and implement.

The approach chosen here is a DEVS. A short overview of the applicability of this approach to the *composed-system* and an overview of executing a DEVS is given next.

4.1.1 Applicability of the DEVS-Formalism

The gentle reader may have asked himself in the previous chapters why so much emphasis has been put on derivation of continuous models and exact (analytical) integration of the dynamic models if, after all, they are only fed into a computer where numerical integration would do almost the same job. The answer to this question is given here by the fact, that it is only possible for exact models to tolerate dynamic[2] time-steps and to provide exact event-prediction capabilities.
In fact, a major criterion for applicability of the DEVSformalism is the suitability of the term *event* and the possibility of *predicting* events.

It is thus conjectured that DEVS is suited very well for specification of the *overall system*, because the system offers distinct events, each effecting a number of components in the system for which the *component-models* offer

[2] varying in size and possibly large

exact one-step-integration in *inter-event-intervals*.
The term *event* hereby applies to the following points in time and causes:

- When a previously rolling ball is switched to passive since its velocity has dropped below the threshold $\vec{v}_{min}$.
- Whenever a robot is due for a controller-step, possibly changing the values v^s and ω which determine its path and sensor-value computations are required.
- When a collision occurs between any objects.
- And finally, when a robot's state is switched from *driving* to *slipping* again after it has collided with another object.

It may now be recognized that models have been given in the last chapters for all components which allow to take *single* integration-steps between the events listed above.
For a correct system-specification or implementation of the overall model, these are the only events which have to be considered. In each, certain actions and computations are to be performed by the effected components.
Another event-type is implemented for the simulator, triggering the output to be observed, but which is not essential to the simulation and is thus not described here.

4.1.2 Overview of DEVS-Execution

Execution of a DEVS that is to say its simulation, is in principle straightforward: The Simulator holds a variable *time* which represents simulation-time and is initialized with 0. For every simulation-step, each component c_i *schedules* itself for the time of its next event t^{next} (i.e. predicts when an event will occur). The simulator collects all predictions and chooses the minimum of those times $min\{t_i^{next}\} \geq time$. At this point the simulator may advance *time* to just this value, since no other event will occur earlier. With the advanced time, the simulator executes a function of all component(s) which have scheduled themselves for that time, handling the corresponding events (components that have not scheduled at that time and which are not effected by other components are not executed).
Whenever a component's function is executed in this manner, it may exchange messages with other components and thus (possibly) influence others. Therefore *scheduling* here really means *predicting*, stressing the fact that it may be possible for a component c_j to predict the time of its next event, while some other component c_k did also for an earlier time and is thus executed before c_j and may influence c_j in a way, such that its prediction has to

be discarded and it is executed prematurely, possibly *rescheduling* itself for another time.

4.2 The DEVS-Formalism

The DEVS-formalism is rooted in system-theory and provides a rigorous formalization and foundation for well-characterized system descriptions. Since its introduction in the seventies, it has gained wide acceptance and has experienced a number of further developments and extensions.
Two formalisms will be presented of which the first is a system-specification framework which can be applied to the single components (such as balls, robots and the guard) and the second is a framework for integrating several components into a *coupled model* which will provide a formalization of the overall model of the world.
Static objects do not need to be modeled as a DEVS-component, since they don't experience any kind of event (i.e. their state does not change).

4.2.1 Classic DEVS with Ports

Definition 4.2.1. A (Classic) Discrete Event System Specification (DEVS) with Ports is a structure DEVS $= \langle S, X, Y, \delta^{int}, \delta^{ext}, \lambda, \tau \rangle$, where

- S is the set of states,
- $X = \{(p, v) \mid p \in InPorts; v \in X_p\}$ is a set of input-ports and values,
- $Y = \{(p, v) \mid p \in OutPorts; v \in Y_p\}$ is the set of output-ports and values,
- $\delta^{int} : S \to S$ is the internal transition function,
- $\delta^{ext} : Q \times X \to S$ is the external transition function, where
  $Q = \{(s, e) \mid s \in S; [0, \tau(s)] \ni e$ denotes the elapsed time since the last state-transition$\}$ is called the set of total states
- $\lambda : S \to Y^*$ is the output function,
- $\tau : S \to \mathbb{R}_0^+ \cup \{\infty\}$ is the time advance function.

The semantics are given by the following:
A DEVS-representation of a component model (from now on simply referred to as a DEVS) adopts the notion of *state*, which includes all information about the component necessary to determine its trajectory in *state-space*

while no interaction with other components take place (the set of states is thus given by all values of variables already used in the models derived in Chapter 2 and 3.)
The input and output sets X and Y serve as a base for message-exchange with other components through ports. There are input-ports and output-ports, where an output-port of one component will be *coupled* with an input-port of another component. The coupling between all components will be taken care of by the Coupled Model (CM) presented in the next subsection.
The interesting semantics of the formal DEVS-model is given by the functions δ^{int}, δ^{ext}, λ and τ:

- The time-advance function $\tau(s)$ determines how long a component is to remain in a state $s \in S$ if nothing "special" happens (i.e. no event occurs). In terms of the developed simulator, τ is to be interpreted as the component's prediction when its next event will take place.

- The output-function λ is invoked whenever $\tau(s)$ time-units have elapsed after the last state-transition. It then produces an output $\lambda \in Y^*$, which is to be interpreted as a (possibly empty) set of messages which are sent out over their corresponding output-ports.

- The internal-transition function $\delta^{int}(s)$ is always executed whenever $\lambda(s)$ has just been invoked. It determines the next state $s' = \delta^{int}(s)$ of the component, (then, $\tau(s')$ is the prediction of time spent in this new state s').

- The external-transition function $\delta^{ext}(s, e, x)$ is invoked whenever the component receives a message on one of its input-ports from another component, where s is the current state just as in the internal-transition function, $e \in \mathbb{R}_0^+$ is the amount of time that has been spent in s and $x \in X$ is the message received from that component.
  As a result of invoking δ^{ext} the component's next state then is $s' = \delta^{ext}(s, e, x)$.

The DEVS-formalism presented here is a slightly modified version of *Classic DEVS with Ports*, as presented in [ZPK00] and is an extension of the original definition of a DEVS in [Zei76].
The formalism is modified by allowing multiple outputs per invocation of the output-function λ. This would, in general, require handling of situations where a component receives several messages at once, but which is not necessary here since the actual DEVS-models in the simulator will never sent out more than one message to another model (multiple messages *are* sent, but each to a *different* model).

4.2.2 A Coupled Model of DEVS-Models

Definition 4.2.2. A Coupled Model (of several DEVS) is a structure CM = $\langle D, \mathcal{C}, \Sigma\rangle$, where

- D is the set of components coupled by the model, where each component $d \in D$ is a $\text{DEVS}_d = \langle S_d, X_d, Y_d, \delta_d^{int}, \delta_d^{ext}, \lambda_d, \tau_d\rangle$
- $\mathcal{C} = \{((a, p_a), (b, p_b)) \mid a, b \in D; p_a \in OutPorts_a; p_b \in InPorts_b\}$ is the coupling-relation, specifying the coupling of the DEVS-components,
- $\Sigma\colon 2^D \setminus \{\,\} \to D$ is the tie-breaking (select) function.

The CM-formalism given here is an enormously simplified version of the *Classic DEVS Coupled Model* as introduced in [ZPK00], where originally it was defined such that it could serve as a DEVS itself to be included in another coupled model in a hierarchical manner. This capability is not needed here.
The semantics are very easy: Each DEVS d is executed as described in subsection 4.2.1. Whenever d produces an output y (over output-port p_y) the receiver of that output is looked up in the coupling-relation and is fed the value of the output over its corresponding input-port (i.e. its δ^{ext}-function is executed with this value).

The tie-breaking function Σ serves the purpose of resolving conflicts whenever several DEVS-models have scheduled themselves for the same point in time (i.e. the λ and δ^{int} functions of several components are to be executed). For example, if $\hat{D}$ is the set of components which wait for invocation of their functions at the same time, then $d^* = \Sigma(\hat{D})$ is the component which will be executed first (or rather its δ^{int}-function).

It is also noteworthy that, if the tie-breaking function chooses one component d^* to be executed first, the external transition functions of any receivers of d^*'s output will immediately be executed also (before their internal-transition function is executed if they happen to be one of the components in $\hat{D}$).

4.2.3 Summary of the DEVS-Formalism and Coupled Model

To summarize the (so far very) abstract formalism and its applicability to the world-model it is recaptured that all components in the world can be modeled as a DEVS with ports where the set of states represents all information about the component.

In simulating the world-model (i.e. robots acting in a dynamic environment) it is of interest how the states of the components change over time (e.g. what paths robots drive in physical space in relation to other objects and any other information).
Since all components are to be modeled as a DEVS, state-changes may only occur whenever an event takes place. Relative to a component the DEVS-formalism classifies events to be either *internal* or *external* (corresponding to the functions δ^{int} and δ^{ext}), where internal events of one component may invoke external events of another.

Any change in any component thus has to originate from an internal event of a component (possibly from itself).
The list of all possible events in the coupled model of the world given in section 4.1.1 on page 33 can thus be differentiated by distinguishing between external and internal events for components:

	ball	**robot**	**guard**	**display**
passive	internal		external	
stopslip		internal	external	
contstep		internal	external	
collision	external	external	internal	
display	internal	internal		external

For each type of event in the world, the table above shows all possible sources (components with internal events) and all component-classes that may (possibly) be effected by that type of event (through invocation of external events).

Illustrations of the two event types which should clarify the DEVS-semantics can be found in Figures 4.1 and 4.2 on page 39.
Both figures (schematically) display the states of a component in the state-set S, its inputs in X and outputs in Y. In the illustration of X the elapsed time e is also included although, strictly speaking, it is not an *input* to the component (but it is also provided as an argument to the function δ^{ext}).

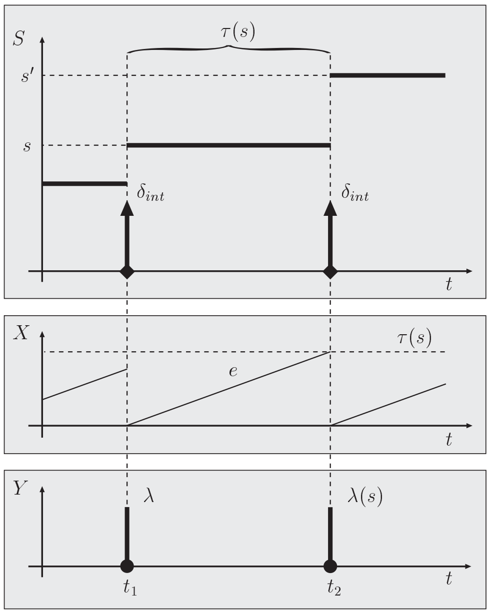


Figure 4.1: At time t_1 the displayed component's *internal* transition function δ^{int} is invoked, yielding a new state s. Since no external events occur, the component remains in that state until, at time t_2, the elapsed time e in that state equals $\tau(s)$. Since $\tau(s)$ is just the amount of time the component was scheduled to stay in s, it's output-function λ is executed, yielding an output $\lambda(s)$. Immediately after sending the output (while still: $time = t_2$), the component's δ^{int}-function is invoked again, yielding a new state $s' = \delta^{int}(s)$.

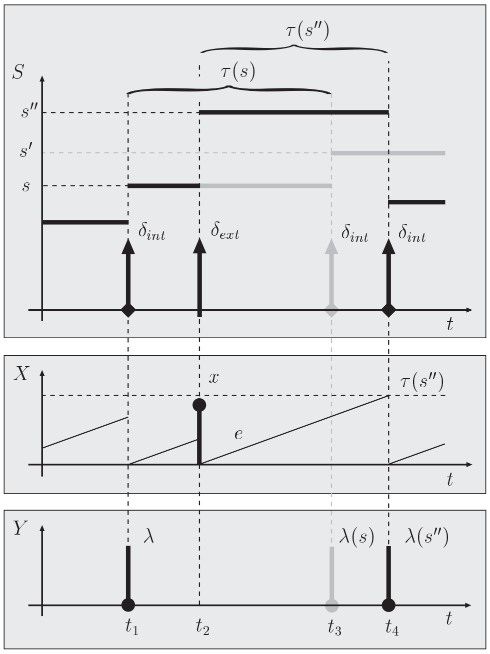


Figure 4.2: At time t_1 the component's internal-transition function δ^{int} is invoked, yielding a new state s just as in the previous figure. This time though, there is an input x coming in from another component at time t_2, causing the *external*-transition function to be executed with arguments $\delta^{ext}(s, e, x)$, where e is the elapsed time spent in s.

Execution of δ^{ext} yields a new state $s'' = \delta^{ext}(s, e, x)$ and a new time to remain in that state $\tau(s'')$. The state-transition $s' = \delta^{int}(s)$ which was scheduled for time t_3 will not take place, but is replaced by a new predicted state-transition $\delta^{int}(s'')$ at time $t_4 = t_2 + \tau(s'')$, where t_4 could be sooner or later than t_3 (depicted is the latter case).

4.2.4 Implementation of the DEVS-Formalism

The actual simulator-implementation, used for all experiments and investigations throughout the thesis, has been done in Java. Details will not be presented to stay within limits and would probably not contribute much to presentation of the principles applied.
Rather than presenting the implementation in a specific programming-language, an object-oriented pseudo-code style will be used to present a concentrate of the algorithms and extract the intrinsic structure of the code. Object-orientation will hereby prove to be a very useful concept especially for DEVS-modeling. The chosen pseudo-code style is C++ and Java-oriented. All keywords are **boldfaced**, methods and fields are specifically declared and method-calls are to be interpreted as *call-by-reference* (which is a characteristic of Java as opposed to C++, where either *call-by-value* or *call-by-reference* is possible). This latter convention makes the code much better readable. Atomic types and methods start with lowercase letters, while complex data-types (Classes) start with uppercase letters.
Besides the usual atomic types of variables found in most languages, existing implementations of classes Msg, MsgSet and StaticObject are assumed, implementation of Msg is such that if msg is an instance of Msg, msg . somevalue extracts the value somevalue from msg (a Msg is an implementation of an input/output-pair $(value, port)$ as specified in Definition 4.2.1.
An implementation of the (abstract) DEVS-formalism itself is given first, from which actual implementations of all components will be derived (in the very object-oriented sense of the word). Two classes are implemented for handling of the DEVS-formalism:
DEVSSimulator and CMSimulator, which implement a DEVS and a composed-model, respectively.

Implementation of DEVSSimulator

A DEVSSimulator is an abstract class (a class to be extended by a subclass) given by Class 4.1 on page 41, which serves as a wrapper-class for implementations of actual (DEVS-) component-models. Implementation of the DEVS-formalism is straightforward:
The four essential DEVS-functions δ^{int}, δ^{ext}, λ and τ are declared abstract, such that extending subclasses have to implement them appropriately. It's basis are the two fields $\mathsf{t}^{\mathsf{last}}$ and $\mathsf{t}^{\mathsf{next}}$ as well as the methods $\mathsf{exec\_}\delta^{\mathsf{int}}()$ and $\mathsf{exec\_}\delta^{\mathsf{ext}}()$.
The fields $\mathsf{t}^{\mathsf{last}}$ and $\mathsf{t}^{\mathsf{next}}$ represent the time of the last state-transition (internal or external) and the time of the next predicted internal event, respectively. The methods mentioned serve as wrappers to the methods δ^{int}, δ^{ext}, λ and τ. They are executed (by a CMSimulator) whenever an internal or external

event occurs for the component.

```
public abstract class DEVSSimulator
private field real t^last, t^next
//reference to the simulator of the composed model.
private field CMSimulator sim
protected field integer index
//abstract methods to be implemented by subclasses.
protected abstract method void δ^int()
protected abstract method void δ^ext(real e, Msg msg)
protected abstract method MsgSet λ()
protected abstract method real τ()

public method constructor(integer i, CMSimulator s) {
      index := i
      sim := s
      t^last := 0
      t^next := t^last + τ()
}
public method real get_t^next() {return t^next}

public method void exec_δ^int(real t) {
      local MsgSet M := λ()
      sim . receive_msgs_from_DEVS(M, this)
      δ^int()
      t^last := t
      t^next := t^last + τ()
}
public method void exec_δ^ext(Msg msg, real t) {
      δ^ext(t − t^last, msg)
      t^last := t
      t^next := t^last + τ()
}
end class.
```

Class 4.1: DEVSSimulator

Implementation of CMSimulator

CMSimulator is a final class which implements the coupled-model of all DEVS-models in the world, which are instances of DEVSSimulator (or rather instances of subclasses thereof). It is the actual top-level class which simulates the world-model. It's implementation given by class 4.2 on page 43 is fairly self-explanatory.
It holds (as fields) references to all objects in the world and controls simulation-time represented by the field *time*. The field C represents the coupling-relation of the coupled-model just as specified in Definition 4.2.2.
In its constructor, it also creates instances of all other objects present in the

world.
The basic simulation loop is located in method run(), where simulation-time is advanced according to the component's predictions and imminent components are executed (their wrapper-function for internal events is invoked).
Implicitly, the *select*-function is implemented in this method, also. If more than one component has predicted an internal event for the minimum of all prediction-times, the component with the minimum *index* is executed. Since all components have been instantiated in that order, the order of priority is: balls, robots and guard (static-objects and the display don't have internal events).
The method receive_msgs_from_DEVS(...) serves the purpose of distributing messages sent out by components to the receivers of those messages.

The Coupling-Relation C

The coupling-relation C is a field of class CMSimulator. Rather than specifying it in formal notation or pseudo-code, it is illustrated in Figure 4.3.

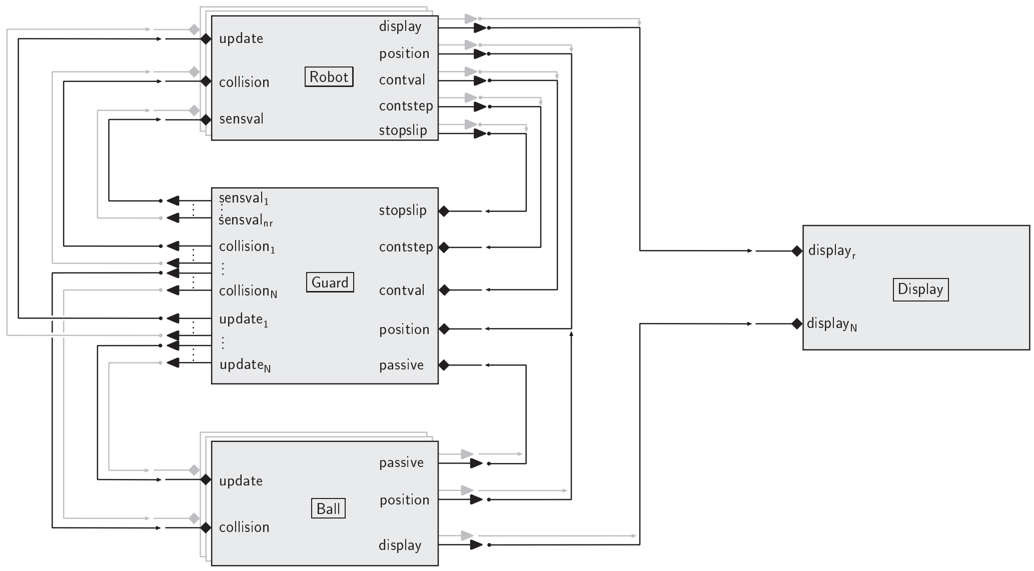


Figure 4.3: Illustration of the coupling relation C. With the exception of Guard and Display, every DEVS-model has a number of input ports connected to exactly one output port of another component and vice-versa.
The DEVS-models Guard and Display have input-ports which receive messages from a whole collection of DEVS-models, where an *index* of the sender is therefore included in each message sent over the corresponding connection.

```
public final class CMSimulator
  private field real time
  //array of static objects.
  private field StaticObject[] S
  //array of DEVSSimulator component models.
  private field DEVSSimulator[] D
  //coupling relation as specified in the component model specification.
  private field set C

  public method constructor() {
      local integer n, n_b, n_r, n_s
      time := 0
      n_b := number of balls
      n_r := number of robots
      n := n_b + n_r + 2
      n_s := number of static objects
      for (i := 1) to n_s {
          S[i] := new StaticObject(i,this,geometry of static object i)
      }
      for (i := 1) to n_b {
          D[i] := new Ball(i,this,initial condition of ball i)
      }
      for (i := n_b + 1) to (n_b + n_r) {
          D[i] := new Robot(i,this,initial condition of robot i)
      }
      D[n − 1] := new Guard(this,initial cond. of D[1]..D[n_b + n_r],geometry of S[1]..S[n_s])
      D[n] := new Display(geometry of S[1]..S[n_s])

      establish coupling relation C between Components D[1]..D[n]
  }
  public method void run() {
      local integer i*
      //infinite loop...
      loop{
          //advance time to next event
          time := min_{i=1..n} {D[i].get_t^next()}
          //get the component which is imminent at that time, if this applies
          //to several components, choose the one with the smallest index.
          i* := min_{i=1..n} {i | D[i].get_t^next() = time}
          D[i*].exec_δ^int(time)
      }
  }
  public method void receive_msgs_from_DEVS(MsgSet M, DEVSSimulator sender) {
      local set R
      local DEVSSimulator r
      for all (msg ∈ M) {
          R := {d | ((msg.port, sender), (p_d, d)) ∈ C; p_d ∈ InPorts_d}
          for all (r ∈ R) do r.exec_δ^ext(msg, time)
      }
  }
end class.
```

Class 4.2: CMSimulator

4.3 The DEVS-Models

The implementations of the DEVS-formalism, given by classes DEVSSimulator and CMSimulator, serve as a basis for implementation of the actual DEVS-models.
Formal DEVS-models will not be given, but rather the implementations serve as implicit DEVS-models in faith of Definition 4.2.1.
Hereby the *state-set* S is just given by the fields of the classes, the state-transition functions δ^{int} and δ^{ext} are given in pseudo-code style and act on the fields, thus effectively performing *state-transitions*. The input and output-sets X and Y are implicitly given by the implementations of methods $\delta^{ext}(\ldots)$ and $\lambda()$, respectively.
For faithful representation of the DEVS-models by their corresponding classes it is to be ensured that the methods $\lambda()$ and $\tau()$ do not modify any fields (since they do not map into S in the original definition).
An overview of implemented classes and their relation in the overall context is displayed in Figure 4.4.

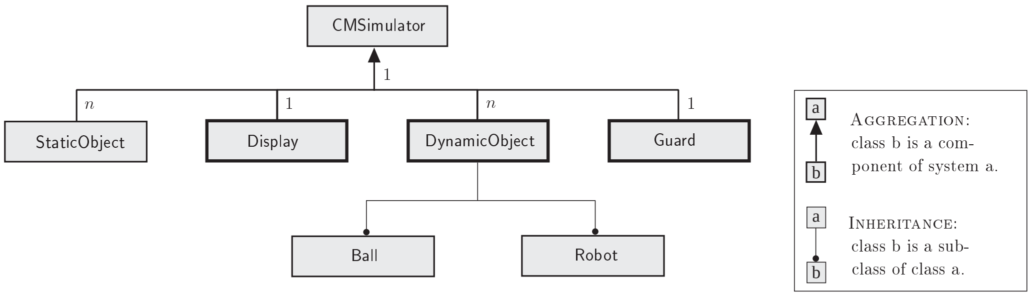


Figure 4.4: Illustration of the essential class-structure of the simulator. Classes with a bold frame are subclasses of DEVSSimulator.

4.3.1 Class DynamicObject

The class DynamicObject, given by Class 4.3, serves as a superclass for the ball and robot DEVS-models; it basically implements the state-set of a dynamic-object, where some fields will not be used depending on whether the derived class represents a robot or a ball and a few update...-methods which serve as wrappers for the dynamic models derived in Chapters 3 and 4.
Furthermore, a set of variables with subscript $_{\mathsf{disp}}$ are introduced which are used for handling *display*-events; similarly variables with subscript $_{\mathsf{evt}}$ represent the predictions for internal events *passive* or *stopslip* (for a ball or robot, respectively).
Variables t with various sub- and superscripts are used for keeping record

of points in time when meaningful events took place, variables $\triangle\mathsf{t}$ serve the purpose of updating t.
Similarly, variables τ and $\triangle\tau$ (with various sub- and superscripts) serve the purpose of predicting future events.
These conventions hold for all DEVS-models.

public abstract class DynamicObject **extends** DEVSSimulator
protected field vector2d $\mathbf{x}, \mathbf{v}$
protected field real $\theta, \mathsf{v}^{\mathsf{s}}, \omega$
protected field real $\mathsf{t}_{\mathsf{upd}}, \mu_{\mathsf{f}}$
protected field string phase //phase $\in$ {passive, slipping, driving}.
protected field boolean send_position
//variables, needed for displaying the object (i.e. its position).
protected field vector2d $\mathbf{x}_{\mathsf{disp}}, \mathbf{v}_{\mathsf{disp}}, \mathbf{x}^{\mathsf{last}}_{\mathsf{disp}}$
protected field real $\theta_{\mathsf{disp}}, \triangle\tau_{\mathsf{disp}}$
//variables, needed for internal events.
protected field vector2d $\mathbf{x}_{\mathsf{evt}}, \mathbf{v}_{\mathsf{evt}}$
protected field real $\triangle\mathsf{t}_{\mathsf{upd}}, \triangle\tau_{\mathsf{evt}}$
//———————————————————————————————————
public method constructor(**integer** i, CMSimulator s) {
    $\triangle\mathsf{t}_{\mathsf{upd}} := 0$
    send_position := **false**
    set_variables_for_display_at_current_position()
    superclass . constructor(i, s)
}
protected method void update_at(**real** t) {
    update_by($\mathsf{t} - \mathsf{t}_{\mathsf{upd}}$)
    $\mathsf{t}_{\mathsf{upd}} := \mathsf{t}$
}
protected method void update_by(**real** $\triangle$t) {
    if (phase = rolling) **then** update_rolling($\triangle$t)
    else if (phase = driving) **then** update_driving($\triangle$t)
    else if (phase = slipping) **then** update_slippinng($\triangle$t)
    $\mathsf{t}_{\mathsf{upd}} \mathrel{+}= \triangle t$
}
private method void update_rolling(**real** $\triangle$t) {
    update $\mathbf{x}$ according to equation (2.1.7) with $\vec{x}_0 = \mathbf{x}, \vec{v}_0 = \mathbf{v}, \triangle\mathsf{t}, \mu_{\mathsf{f}}$
    update $\mathbf{v}$ according to equation (2.1.6) with $\vec{v}_0 = \mathbf{v}, \triangle\mathsf{t}, \mu_{\mathsf{f}}$
}
private method void update_driving(**real** $\triangle$t) {
    update $\mathbf{x}$ according to equation (2.1.16) with $\vec{x}_0 = \mathbf{x}, \theta_0 = \theta, \omega, \mathsf{v}^{\mathsf{s}}, \triangle\mathsf{t}$
    update θ according to equation (2.1.15) with $\theta_0 = \theta, \omega, \triangle\mathsf{t}$
}
private method void update_slippinng(**real** $\triangle$t) {
    update $\mathbf{x}$ according to equation (3.3.8) with $\vec{x}_0 = \mathbf{x}, \theta_0 = \theta, \omega, \mathsf{v}^{\mathsf{s}}, \triangle\mathsf{t}$
    update θ according to equation (3.3.9) with $\theta_0 = \theta, \omega, \triangle\mathsf{t}$
}
⋮

⋮ //———————————————————————————

protected method void update_variables_for_next_display_event(**vector2d** $\hat{\mathbf{x}}$) {

$\mathbf{x}_{\mathsf{disp}}^{\mathsf{last}} := \hat{\mathbf{x}}$

$\triangle\tau_{\mathsf{disp}} :=$ depending on display-resolution and desired smoothness...

update $\mathbf{x}_{\mathsf{disp}}, \mathbf{v}_{\mathsf{disp}}, \theta_{\mathsf{disp}}$ by $\triangle\tau_{\mathsf{disp}}$ just as in update_by(), with $\vec{x}_0 = \mathbf{x}_{\mathsf{disp}}^{\mathsf{last}}, \vec{v}_0 = \mathbf{v}_{\mathsf{disp}}$

}

//———————————————————————————

protected method void set_variables_for_display_at_current_position() {

$\mathbf{x}_{\mathsf{disp}} := \mathbf{x}$

$\theta_{\mathsf{disp}} := \theta$

$\mathbf{v}_{\mathsf{disp}} := \mathbf{v}$

$\triangle\tau_{\mathsf{disp}} := 0$

}

end class.

Class 4.3: DynamicObject

4.3.2 Class Ball

The implementation of a Ball, given by Class 4.4 specifies a fairly straightforward DEVS-model. It's *state-set* is completely inherited from DynamicObject. There are two basic phases the model can be in: passive and rolling. If the ball is rolling, it's next (meaningful) internal event is predicted in $\triangle\tau_{\mathsf{evt}}$ time-units when the ball turns passive (not considering display and send-out events). If the ball is passive, no event is predicted to take place.
Internal events are of type passive, display or send-out events. External events are either collision or update events, where both are due to messages that are received from the guard.

When the guard has predicted a collision which just takes place, a colliding ball (or a robot) receives a notifying message including position and time of the collision, such that no position-update has to be performed by it.
Update-messages may be received by a ball (or a robot) at any time when the guard requests updated position-data of the component (i.e. for collision-prediction or sensor-value computations).
As a response to such a request, the component immediately outputs its updated values on port position ($\tau(s) = 0$ if s is the resulting state from the incoming request and $\lambda(s) = (\mathsf{position}, (\ldots))$).

```
public final class Ball extends DynamicObject
public method constructor(integer i, CMSimulator s,initial condition) {
    initialize variables x, v, μ_f according to initial condition
    if (||v|| > v_th) then
        phase := rolling
        △τ_evt := compute_time_till_passive()
        update_variables_for_next_state_change_event(△τ_evt)
    else
        phase := passive
        △τ_evt := ∞
        v := 0
    endif
    superclass . constructor( i, s)
}
protected method real τ() {
    if (send_position) then return 0
    else if (phase = passive) then return ∞
    else return min{△τ_disp, △τ_evt}
}
protected method set λ() {
    if (send_position) then return {(position, (index, x, v))}
    else if (△τ_disp < △τ_evt) then return {(display, (index, x_evt))}
    else return {(passive, index), (display, (index, x_disp))}
}
protected method void δ^int() {
    if (send_position) then send_position := false
    else if (△τ_disp < △τ_evt) then
        △t_upd += △τ_disp
        △τ_evt −= △τ_disp
        update_variables_for_next_display_event(x_disp)
    else
        t_upd += (△t_upd + △τ_evt)
        △t_upd := 0
        x := x_evt
        v := 0
        x_disp^last := x_evt
        phase := passive
        △τ_disp := ∞
        △τ_evt := ∞
        send_position := false
    endif
}
⋮
```

```
⋮
protected method void δ^ext(real e, Msg msg) {
    △t_upd := 0
    if (msg.port = update) then
        △τ_evt −= e
        △τ_disp −= e
        send_position := true
        update_by(msg.t − t_upd)
    else if (msg.port = collision) then
        phase := rolling
        x := msg.x_coll
        v := msg.v_coll
        △τ_evt := compute_time_till_passive()
        update x_evt, v_evt by △τ_evt just as in function update_by() in superclass...
        set_variables_for_display_at_current_position()
        t_upd := msg.t_coll
    endif
}
end class.
```

Class 4.4: Ball

4.3.3 Class Robot

Implementation of a robot (see Class 4.5) is very similar to a ball with the noteworthy exception that it is an abstract class.
It extends the *state-set* by a few more variables including $\triangle\tau_{\mathsf{cont}}$ which is used for computing the time of the next internal event in case it is a contstep-event.
For this event, a derived subclass needs to provide an implementation of method controller(), which is declared abstract in Robot (hence, the class is abstract). This technique provides an elegant way of providing the basis for several types of robot-implementations (each possibly implementing a different controller by *overriding* controller()).

The phase of a robot may be driving or slipping. Internal events are of type contstep, stopslip, display or are send-out events. External events are of type update, collision or sensval.
A contstep event is triggered according to an arbitrary schedule of controller-steps[3] when a corresponding message is sent out to the guard which, in turn, responds with a sensval-message, triggering an external event upon which the robot invokes its method controller() yielding new controller-values v^{s} and ω which are then sent out to the guard for new collision-predictions.
External events of type update or collision are handled similarly to the case

[3]Throughout the thesis, only regular controller-steps are used in actual simulations

of a ball.
Three variables, $\triangle\tau_{\mathsf{evt}}$, $\triangle\tau_{\mathsf{disp}}$ and $\triangle\tau_{\mathsf{cont}}$ are used for internal-event predictions where the minimum of the three yields the prediction, except when there are messages to be sent out which is done immediately (as was the case for a ball).

```
public abstract class Robot extends DynamicObject
private field boolean wait_for_sensorval, send_contval
private field integer n_sens
//————————————————————————————————————————————————
protected abstract method void controller()
//————————————————————————————————————————————————
public method constructor(integer i, CMSimulator s,initial condition) {
      initialize variables x, θ, v, v^s, ω, μ_f, n_sens according to initial condition
      phase := driving
      wait_for_sensorval := false
      send_contval := false
      △τ_cont := time till first controller-step
      △τ_evt := ∞
      superclass . constructor( i, s)
}
protected method void δ^int() {
      if (send_position) then send_position := false
      else if (send_contval) then
            send_contval := false
            wait_for_sensorval := false
            update_variables_for_next_display_event(x^last_disp)
      else if (△τ_cont < min{△τ_disp, △τ_evt}) then
            if (△τ_evt < ∞) then △τ_evt −= △τ_cont
            △τ_cont := depending on the schedule for controller-events...
            wait_for_sensorval := true
      else if (△τ_disp < △τ_evt) then
            △t_upd += △τ_disp
            △τ_cont −= △τ_disp
            if (△τ_evt < ∞) then △τ_evt −= △τ_disp
            update_variables_for_next_display_event(x_disp)
      else
            △τ_cont −= △τ_evt
            t_upd += (△t_upd + △τ_evt)
            △t_upd := 0
            △τ_evt := ∞
            phase := driving
            x := x_evt
            v := 0
            update_variables_for_next_display_event(x_evt)
      endif
}
   ⋮
```

```
⋮
protected method real τ() {
    if (send_position ∨ send_contval) then return 0
    else if (wait_for_sensorval) then return ∞
    else return min{△τ_cont, △τ_disp, △τ_evt}
}
protected method set λ() {
    if (send_position) then return {(position, (index, x, θ, v))}
    else if (send_contval) then return {(contval, (index, v^s, ω))}
    else if (△τ_cont < min{△τ_disp, △τ_evt}) then return {(contstep, index)}
    else if (△τ_disp < △τ_evt) then return {(display, (index, x_disp, θ_disp))}
    else return {(stopslip, (index, x_evt)), (display, (index, x_evt, θ))}
}
protected method void δ^ext(real e, Msg msg) {
    △t_upd := 0
    if (msg . port = update) then
        △τ_cont −= e
        △τ_disp −= e
        if (△τ_evt < ∞) then △τ_evt −= e
        send_position := true
        update_by(msg . t − t_upd)
    else if (msg . port = collision) then
        phase := slipping
        x := msg . x_coll
        v := msg . v_coll
        △τ_evt := predict time in phase slipping according to equ. 3.3.7...
        compute var. x_evt, v_evt with △t = △τ_evt just as in method update_slippinng...
        set_variables_for_display_at_current_position()
        t_upd := msg . t_coll
    else if (msg . port = sensval) then
        //function controller(...) may set v^s and ω according to sensor-values
        controller(msg . v_sens[1]..msg . v_sens[n_sens])
        send_contval := true
    endif
}
end class.
```

Class 4.5: Robot

4.3.4 Class Guard

Guard is the last implementation to be presented (see Class 4.6). It is the most complex one since it handles interaction among all other components. It's state includes information about the state of all static as well as all dynamic objects in the world since this information is needed for collision-prediction and sensor-value calculation (although these variables may not be up to date at all times).
The guard's only possible internal event (besides send-out events) is collision. Whenever a collision takes place the collidees receive corresponding messages over the collision-port, while all other *non-passive* components receive update-events.
Information about the components states are held in *index-sets* which are denoted by an I with some subscript (e.g. I_{pass} denotes the set which includes all indices of balls that are currently passive).
In a simulation-run the guard only comes into play whenever a collision occurs (as described above) or when a robot takes a controller-step. In the latter case, the guard receives a contstep-message whereupon it sends out update-messages to all *other* components which are not passive and did not emit a message at the same point in time. It then waits for all incoming new position-messages from those components, such that sensor-values can be computed and sent back to the robot, thereafter waiting for a contval-message from each robot which took a controller-step, to update the collision-predictions.
In the pseudo-code version of class Guard, a readily implemented method compute_next_collision() is presupposed which, besides predicting the collision times, also computes the positions of the colliding components.

```
public final class Guard extends DEVSSimulator
  private field integer N_contval, N_obj, N_update, n_msg
  private field boolean wait_for_contval, wait_for_position, send_sensval, send_update
  private field real △τ_coll, t
  private field vector2d[] x, v
  private field real[] θ, v^s, ω
  private field vector2d[][] x_static
  private field integer [] n_sens
  private field real[][] v_sens
  private field set I_all, I_coll, I_cont, I_pass, I_stop, I_robot, I_slip // indexsets
  private method void received_all_requested_updates() {
      if (wait_for_contval) then
          compute_sensor_values()
          send_sensval := true
      else if (wait_for_position) then compute_next_collision()
      wait_for_position := false
  }
  ⋮
```

```
⋮
public method constructor(integer i, CMSimulator s,initial conditions of all objects) {
      N_obj := number of dynamic objects (balls and robots)
      for (i := 1) to N_obj do init vars x[i], θ[i], v[i], v^s[i], ω[i] acc. to initial conditions
      initialize array x_static according to initial conditions
      I_all := {1, 2, .., N_obj}
      I_pass := { i | object i is a ball which is passive}
      I_robot := { i | object i is a robot}
      I_coll := I_cont := I_stop := I_slip := { }
      n_msg := 0
      t := 0
      compute_next_collision()
      superclass.constructor(i, s)
}
protected method real τ() {
      if (send_sensval ∨ send_update) then return 0
      else if (wait_for_contval ∨ wait_for_position) then return ∞
      else return △τ_coll
}
protected method set λ() {
      local real t_coll := t + △τ_coll
      if (send_update) then return {(update_i, t) | i ∈ I_all \ (I_pass ∪ I_stop)}
      else if (send_sensval) then return {(sensval_i, v_sens[i]) | i ∈ I_cont}
      else return {(collision_i, (t_coll, x_coll[i], v_coll[i], impulse[i])) | i ∈ I_coll} ∪ ...
      ... {(update_i, t_coll) | i ∈ I_all \ (I_pass ∪ I_coll)}
}
protected method void δ^int() {
      if (send_sensval) then send_sensval := false
      else if (send_update) then
            send_update := false
            if (wait_for_contval) then N_contval := |I_cont|
            N_update := (N_obj − |I_pass ∪ I_stop|)
            if (N_update = 0) then received_all_requested_updates()
            I_stop := { }
      else
            I_pass \= I_coll
            I_slip ∪= (I_coll ∩ I_robot)
            N_update := (N_obj − |I_pass ∪ I_coll|)
            t += △τ_coll
            if (N_update = 0) then compute_next_collision()
            else wait_for_position := true
            for all (i ∈ I_coll) {
                  x[i] := x_coll[i]
                  v[i] := v_coll[i]
            }
      endif
}
⋮
```

```
⋮
protected method void δ^ext(real e, Msg msg) {
    t += e
    if (msg.port ∈ {stopslip, passive}) then
        send_update := true
        wait_for_position := true
        if (msg.port = passive) then I_pass ∪= {msg.index}
        else if (msg.port = stopslip) then
            I_slip \= {msg.index}
            I_stop ∪= {msg.index}
        endif
        x[msg.index] := msg.x_evt
        v[msg.index] := 0
    else if (msg.port = contstep) then
        if (not wait_for_contval) then
            send_update := true
            wait_for_contval := true
            I_cont := { }
        endif
        I_cont ∪= {msg.index}
    else if (msg.port = position) then
        n_msg += 1
        x[msg.index] := msg.x
        v[msg.index] := msg.v
        if (msg.index ∈ I_robot) then θ[msg.index] := msg.θ
        if (n_msg = N_update) then
            n_msg := 0
            received_all_requested_updates()
        endif
    else
        n_msg += 1
        v^s[msg.index] := msg.v^s
        ω[msg.index] := msg.ω
        if (n_msg = N_contval) then
            n_msg := 0
            compute_next_collision()
        endif
    endif
}
end class.
```

Class 4.6: Guard

4.3.5 Interaction between the DEVS-Models

After presenting the (pseudo-) code of the implementation, an overview of the message-flow is given in Figure 4.5.
In Figure 4.6 and 4.7 on page 55 typical event-schedules are given in a similar style as the DEVS-formalism was illustrated in Figures 4.1 and 4.2 on page 39. Both figures schematically illustrate how events effect the models state. In general, there are several different types of interactions in combined discrete-continous simulation models. In [Kli96] they are categorized according to the following three basic types:

(i) A discrete event changes the value of a continuous variable

(ii) A discrete event affects the relation governing a continuous variable

(iii) A continuous variable reaching a threshold causes a discrete event

In the simulation, all basic events are caused by a type (iii) interaction (e.g. a ball turns *passive* when its velocity has reached a threshold, a *collision* takes place when time has advance to t_{coll}, etc.).
All basic events cause, in turn, interactions of type (i) and (ii) (e.g. a ball's velocity is set to $\vec{0}$ upon a *passive*-event, affecting the relation governing its position, a robot's drive is set according to new controller-values upon a *contstep*-event, also affecting the relation governing its position,...).

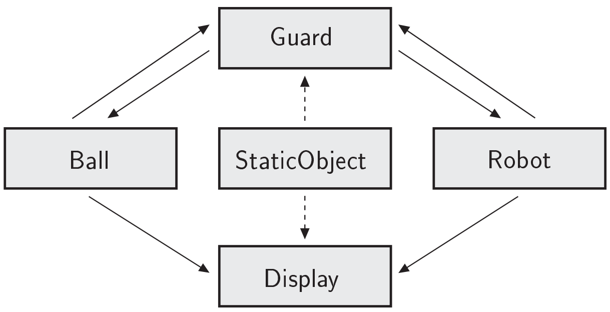


Figure 4.5: Illustration of the message-flow (arrows) between the components in the simulator. Messages between robots and balls are never directly exchanged, but interaction always takes place indirectly. Static objects don't participate in message-flow, their initial condition is only once sent to the guard and display (dashed arrows).

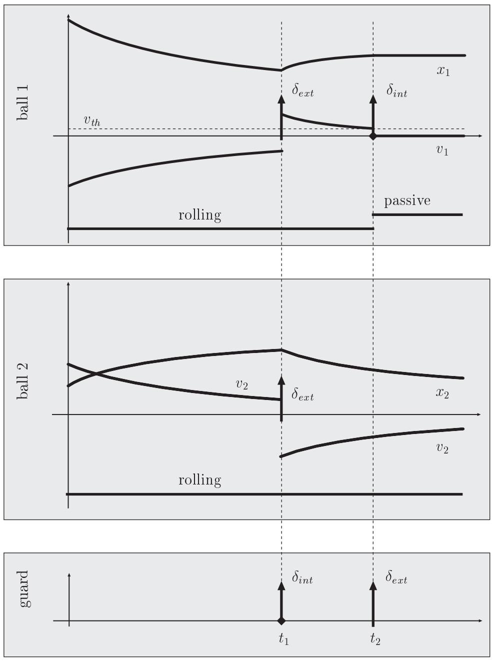


Figure 4.6: A typical event-schedule is schematically illustrated for the guard and two balls. The guard has predicted a *collision* between ball 1 and ball 2 at time t_1, thus a message is sent out to both at t_1, invoking their *external*-transition function. Velocities are changed according to collision handling and both have to recompute their predictions for next internal (*passive-*) events.
At time t_2 ball 1 turns passive, setting its velocity to 0 and notifying the guard which, in turn, has to update its collision predictions accordingly.

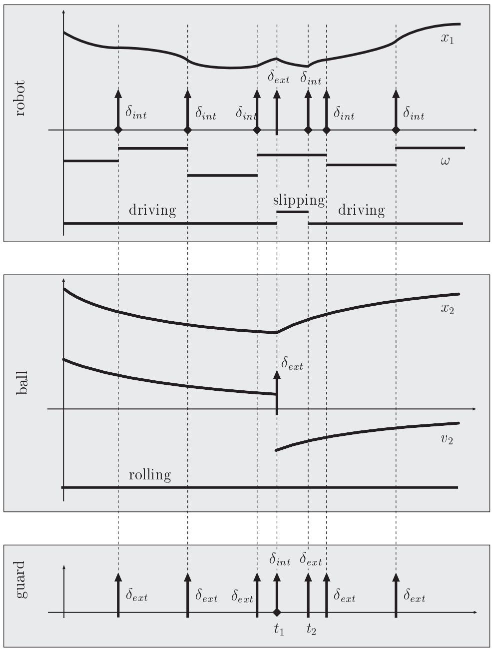


Figure 4.7: A typical event-schedule for the guard, a robot and a ball is shown. The robot regularly takes controller-steps, when its internal-transition function is invoked and a message is sent out to the guard. Subsequent turns in message-flow per event are not shown to avoid cluttering. At time t_1 a *collision* between the robot and the ball takes place, causing the robot to switch to state *slipping* and the ball to change its velocity. At time t_2 the robot switches back to *driving* again, notifying the guard about this event via an emitted message (which is given by the output-function $\lambda()$).

4.4 Summary and Aspects of the Implementation

With the implementations of the DEVS-models a complete simulator has been implemented which applies *event-driven* discretization of the time-domain. Finally, some aspects are summarized and addressed which emphasize the characteristics of this approach and illustrate its power.

4.4.1 Dynamic Scheduling

For displaying (among other activities), dynamic scheduling of the components has been implemented enabling each component to be displayed independently according to its schedule.

This feature provides the possibility of scheduling display-events only when they are needed. For example two dynamic objects a and b which move according to (tangential) velocities v_a and v_b, where $v_a = c\, v_b$, may schedule their display-events such that component a is displayed c times more often per unit time, yielding smooth output yet preserving optimal performance (with the moderation that overhead of the dynamic-scheduling-mechanisms is not considered).

4.4.2 Event-Driven Scheduling

The *event-driven* approach chosen enables the simulator to take time-steps as large as any inter-event-interval, effectively partitioning the time-domain into intervals over which analytically integrable dynamic models for the components have been derived, solving the problem of dealing with adequate numerical methods for integration.

4.4.3 Speculative Computation

A presupposition to the aspects above is the possibility of effectively performing *speculative* computations which provide the foundation for partitioning the time-domain according to events. Speculative computation in the simulator takes place whenever collisions, passive-events or stopslip-events are predicted. These computations are termed *speculative* because they are not guaranteed to take place, but may be discarded as another event takes place earlier (or even concurrently).

Chapter 5

The Notion of Homeokinesis

After development of a complete simulator for simulating dynamic worlds consisting of three basic types of objects, the stage is now set for investigation of what may be called *behavior* of robots embodied in the world and its *emergence*.

5.1 Observing Robots

In any world, the simulator is capable of simulating, robots are the only entities which might exhibit something that would qualify for being called *behavior*.

5.1.1 Behavior

To anybody studying (or working in) the fields of AI or AL, only *meaningful* or *intelligent behavior* is a subject of interest or behavior which *makes sense* in some way and this is just the focus here.
It is clear that no exact objective definition of the term behavior has been given so far, but none will be and probably cannot be given – it remains of rather subjective nature and greatly depends on the judging observer, whether or not a behavior deserves to be called meaningful or intelligent.

As a basis for judging whether or not some observed behavior is intelligent, it is to be considered what options exist. For the robots being simulated, the options in any situation are given by all possible choices of controller-values v^s and ω.
Choices of controller-values are made by robots at all times when a controller-step is invoked. The actual choice will be referred to as an *action*.
Actions may depend on present sensor-values provided to the robot which are termed *perceptions* and possibly a history of past actions and perceptions

or information gained from them, which will be called *experience*.

Starting with these terms, one may reformulate *behavior* as being the relation governing which *action* is chosen for a given *perception* (possibly) depending on past *experience*.

5.1.2 Adaptive Behavior

Since no prior knowledge about its environment (nor about itself) is implanted in the robot in any way, its behavior could not be meaningful if it did not *adapt* in any way.
Adapting here means the act of modifying behavior according to experience. When adaption is performed such that behavior improves over time according to some given performance-measure, the term *learning* applies.

As a consequence of the statements above it becomes clear that for understanding or modeling behavior and adaption or learning, the perspective has to be shifted from that of an external observer taken upon the world to an *internal* perspective of the robot's view of the world while interacting with it, its *perceptions* (this issue is also well known as the *perspective-issue* in the context of the frame-of-reference problem, [PS99, p. 112]).

5.2 The Homeokinetic Principle

In faith of embodied cognitive science, dealing with synthetic modeling of agents for emergence of behavior in the context of agent-environment interaction, a new principle has been introduced in [DSP99], called homeokinesis, which operates by adapting just the *internal* perspective described earlier.
It is a completely unspecific and fairly general principle, but has shown to produce seemingly goal-oriented behaviors for just the kind of setting as considered by the simulator in this thesis.

According to [Der00], the term homeokinesis descends from *homeostasis* which has been found to be a general principle in physiology as early as 1939 and is known within cybernetics at least since the days of Ashby in 1960, stating that reactions of the body (or behavior) can be understood in terms of interwoven control-circuits trying to stabilize certain variables in a stable stationary state or within certain limits (the *viability zone*), thus producing actions in order to uphold this state.
In this context, behavior may be viewed as an *emergent* property, meaning that behavior may be observed which is not build into the parts of the system under observation and may thus be interpreted as a byproduct or

epiphenomenon of the dynamics produced in order to uphold a stationary state.
Homeokinesis is a generalization of homeostasis, aiming at stabilization of variables in a *kinetic regime*, rather than a stationary state.

5.2.1 Controller and Self-Model

In its most general setting, the principle of homeokinesis requires the presence of a *controller* (determining the actions for given perceptions) and a *self-model*, predicting the perceived effects of actions.
More specifically, the self-model is to (correctly) predict what state the robot finds itself situated in after having chosen a particular action in a predecessing state. Hereby, the term state really refers to *perceptions* in that situation (since this is the only available information to the robot about the more general notion of state in the world).
For the homeokinetic principle to be applicable, both units need to be *adaptive*.

Briefly, the controller determines *actions* whereas their effects, resulting from interaction with the environment, are predicted by a self-model.

5.2.2 Learning

In the setting described above, application of homeokinetic control aims at optimizing the controller as well as the self-model with the objective, that the error of the self-model's predictions is to be minimized.

Thus, effects of the resulting control promise to be *predictable* (by the self-model) with respect to the *internal* perspective. Hereby, the self-model defines the kinetic regime that the robot's *perceptions* are stabilized in.
The working-hypothesis of homeokinesis is that *predictable* behavior is also behavior that makes sense or is reasonable.

5.3 Application of Homeokinetic Control

A more specific framework enabling the application of homeokinetic control to the kind of setting considered by the simulator is presented in this section. It has been developed in [DP99, Der00] and serves as a basis for Examples I and IIa given at the end of this chapter, which aim at demonstrating the kind of behavior emerging from homeokinetic control.

5.3.1 Controller

In order to precisely formulate in mathematical terms what has been described in words so far, a simple parameterized controller C is presented of the general form:

$$\omega(t) = C\left(\vec{s}(t), \vec{c}\right), \tag{5.3.1}$$

where the determined *action* at any point in time is restricted to choosing a value for angular-velocity $\omega(t)$, depending on the current sensor-values (*perceptions*) $\vec{s}(t)$ and parameters $\vec{c}$ (for which the dependence on time due to adaption is implicit).
The forward-speed v^s is set to a small (positive) constant v^s_{const} forcing the robot to always keep moving.

Specifically, the controller is a simple formal model of a neuron where the inputs consist of the sensor-values and the weights are the parameters of the controller:

$$\omega(t) = \omega_{max} \tanh\left(\sum_{i=1}^{n_s} c_i\, s_i(t)\right), \tag{5.3.2}$$

where tanh is the (odd) activation-function of the neuron and ω_{max} is the maximum possible angular velocity according to the restrictions imposed by the maximum motor-speeds and reserved capacity for driving-velocity v^s_{const}.
It is assumed, that controller-steps are taken at small regular time-steps.

5.3.2 Self-Model

For mathematical formulation of the self-model, it is useful to adopt a dynamical-systems point of view onto the sensor-values (*perceptions*):
Since the sensor-values at the next controller-step $\vec{s}(t+1)$ are a function of the *action* chosen at the current controller-step as well as current sensor-values $\vec{s}(t)$, the dynamics in *sensor-space* may be given by a discrete-time dynamical system of the following form:

$$\triangle\vec{s}(t) = \vec{s}(t+1) - \vec{s}(t) = W(\vec{s}(t), \omega(t)), \tag{5.3.3}$$

where $\omega(t)$ is the current *action* and W is a (world-) function given by the dynamics of the world (or more specific by the models for all objects in the world and their interaction).
Explicitly including the controller C, this may be reformulated as:

$$\triangle\vec{s}(t) = W\left(\vec{s}(t), C\left(\vec{s}(t), \vec{c}\right)\right). \tag{5.3.4}$$

It is clear, that W generally is a highly stochastic function since the perceptions $\vec{s}$ are not capable of capturing the *overall* state of the world, but on which future perceptions will depend.

In analogy to the above formulation of the dynamics in *sensor-space*, a general formulation of a self-model Φ is:

$$\triangle \vec{s}^{p}(t) = \Phi\left(\vec{s}(t), \omega(t), U\right), \tag{5.3.5}$$

where $\triangle \vec{s}^{p}(t)$ is the *predicted* change in sensor-values at the next controller-step with respect to the current sensor-values and U is a set of parameters. The specific approach taken in this chapter is the same as in [DP99, Der00]:

$$\Phi_i\left(\vec{s}(t), \omega(t), U\right) = \omega(t) \sum_{j=1}^{n_s} u_{ij}\, s_j(t), \tag{5.3.6}$$

or in matrix-notation:

$$\Phi = \omega(t)\, U\, \vec{s}(t), \tag{5.3.7}$$

where $U = (u_{ij})$ is a $(n_s \times n_s)$-matrix of the self-model's parameters and its elements u_{ij} may be interpreted as the additive contribution of the product $\omega(t)s_j(t)$ to the prediction of change in sensor-value $\triangle s_i(t) = s_i(t+1) - s_i(t)$.

This approach will later be referred to as the *bilinear* self-model.
It has been shown in the publications mentioned above, that this type of a self-model is already capable of providing a sufficient basis for emergence of various kinds of nontrivial behaviors (actually this has been shown for even simpler models, namely the trivial self-model $\Phi = \vec{0}$).

These formulations for the controller and the self-model (in equations (5.3.7) and (5.3.2)) will serve as a basis for experiments in different settings using the developed simulator at the end of this chapter.

5.3.3 Adaption of the Controller and the Self-Model

It was said earlier that adaption is to take place, such that the prediction-error is minimized, thus a measure of prediction-error at time-step t is given:

$$E(t) = \frac{1}{2} \sum_{i=1}^{n_s} \left(e_i(t)\right)^2, \tag{5.3.8}$$

where $e_i(t) = \triangle \vec{s}_i(t-1) - \triangle \vec{s}_i^{\,p}(t-1)$ is the error in predicting sensor-value $s_i(t)$ in the current time step, $\triangle \vec{s}_i(t-1)$ is the experienced change in sensor-values wrt. the last time step and $\triangle \vec{s}_i^{\,p}(t-1)$ is its prediction according to self-model Φ as given in the last subsection, hence $E(t) = E(\vec{s}(t), \vec{c}, U)$, such that the controller's parameters $\vec{c}$ as well as the self-model's parameters U

may be learned by gradient-descent, yielding updates (for $\vec{c}$):

$$\triangle c_i(t) = -\eta_c \frac{\partial}{\partial c_i} E(t) = -\eta_c \frac{\partial E}{\partial \omega} \frac{\partial \omega}{\partial c_i}, \tag{5.3.9}$$

where η_c is the *learning-rate* of the controller and the second partial derivative may be computed using equation (5.3.2).
The partial derivative $\frac{\partial E}{\partial \omega}$ is approximated by adding a small periodic oscillation $\xi(t)$ to $\omega(t)$, such that the new value $\hat{\omega}(t)$ is given by:

$$\hat{\omega}(t) = \omega(t) + \xi(t) = \omega(t) + a \sin\left(\frac{b\,t}{2\pi}\right), \tag{5.3.10}$$

where $\omega(t)$ is given by equation (5.3.2) with a modified value of ω_{max} such that, additionally, the value $\xi_{max} = a$ is considered, where a is small and denotes the amplitude and b is the (large) period of the oscillation. By "large period" it is meant that the period is to be chosen such, that the difference in the additional term ξ between successive controller-steps is small.

By stating, that $E(\vec{s}, \vec{c}, U) = E(\vec{s}, \hat{\omega}, U)$, the partial derivative $\frac{\partial E}{\partial \omega}$ may be obtained by a time-average over $\xi(t-1)E(t)$ (the details of this "trick" are given in [DP99]).
The update-rule given by equation (5.3.9) thus evaluates to:

$$\triangle c_i(t) = -\eta_c\, s_i(t-1) \left(1 - \tanh^2\left(\sum_{j=1}^{n_s} c_j\, s_j(t-1)\right)\right) \xi(t-1) E(t). \tag{5.3.11}$$

The update-rule for the self-model can be derived directly from the self-model, given by equation (5.3.6) and the prediction-error, given by equation (5.3.8):

$$\triangle u_{ij}(t) = \eta_u\, e_i(t)\, \omega(t \hat{-} 1)\, s_j(t-1), \tag{5.3.12}$$

where η_u is the learning-rate parameter for the self-model.

5.4 Experiment I

Since homeokinesis has been introduced, the stage is now set for a first experiment to investigate resulting behavior (if it may be called so) of robots in a dynamic environment.

The world in this experiment will only be populated by dynamic objects. Its borders obey periodic boundary conditions (i.e. the borders are *wrapped around*), it thus effectively mimics a world of infinite size.
Specifically the setting consists of 5 balls and 8 robots, of which 6 robots are driven by homeokinetic control as introduced in the previous setting. The other two remaining robot's controllers are *obstacle-avoiders* with a fixed controller very much like a Braitenberg-vehicle (e.g. see [PS99, pp. 181-198]), but with three sensors on each side. All robots drive at the same speed (disregarding angular velocity).
All homeokinetic robots are of exactly the same type except for their size and mass, which varies slightly, all are equipped with a total of only 4 sensors.

5.4.1 Emerging Behaviors

Given the simple setting of this experiment, quite a few interesting behaviors emerge while the controller's parameters evolve. Namely, all (homeokinetic) robots *learn* to approach any type of object within their field of view. Effectively (as a result of the object's interactions), this results in behaviors which may be categorized as:

- object *transportation* in case the approached object is a ball and
- object *persuasion* in case the approached object is another robot and is thus moving itself.

Some snapshots of the experiment are given in Figure 5.1, where numerous variations and combinations of these two basic types of behaviors can be observed for *all* homeokinetic robots in the experiment, which thus seems to be a stable attractor in the controller-predictor dynamics for the given world. (All conducted similar experiments with slightly varied initial conditions and learning-rate parameters also yield very similar results).

5.4.2 Evolution of the Parameters

A parameter-evolution for the self-model and controller for one robot (other robot's parameters evolved quite similar) are given in Figure 5.2 and 5.3, respectively.

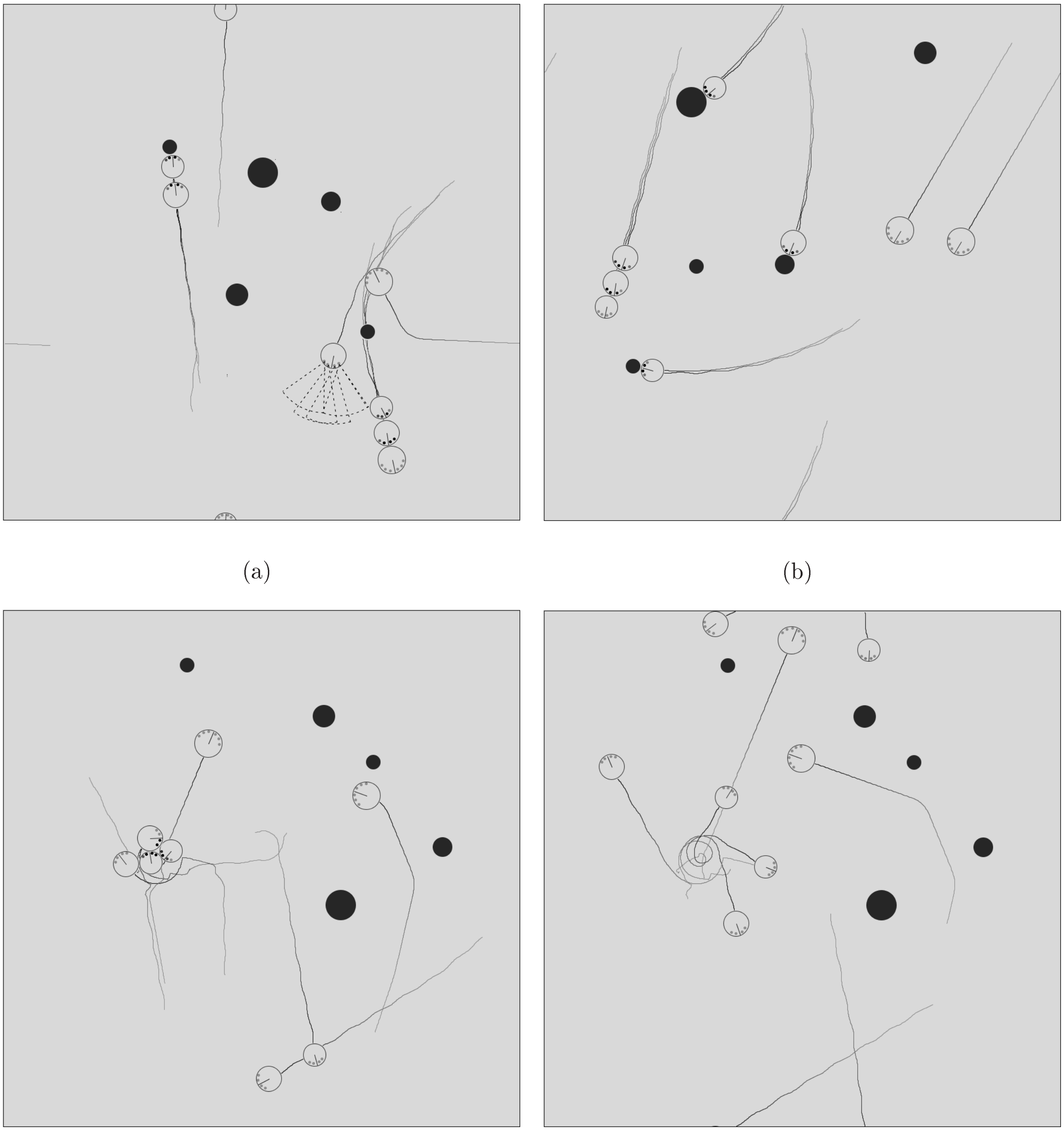
Recalling the sensor's numbering from Figure 2.6 and looking into the evolution of parameters it may be realized that values u_{ij} for *adjacent* sensors evolve to especially *high* absolute values, reflecting the fact that for a given angular velocity ω a change in a sensor's value may most likely be well predicted by the adjacent sensors (especially the one in the direction it is turning to).

In accordance with this observation it may furthermore be observed that all such parameters corresponding to adjacent sensors have *positive* values if they serve the purpose of predicting the sensor to the right (wrt. Figure 2.6) and *negative* values otherwise.
For the controller it may be observed that the outermost sensors are the dominant ones (in terms of absolute value), causing the robot to turn towards perceived objects.

5.4.3 A first Conclusion

In this first experiment, the principle of homeokinesis has shown to live up to expectations. In [DSP99, DP99] it was claimed that homeokinesis may serve as a general principle for the self-organization of robot behavior inducing specific, seemingly goal-oriented, behaviors in a complex external world and that basic behaviors can be acquired by an autonomous agent without the help of externally defined fitness functions or learning rewards (such as in evolutionary computation or reinforcement learning). This could be reproduced to full extent.

The experiment also illustrates very well, how complex behavior (observed from an external point of view) can be rooted in simple internal mechanisms, where the observed complexity arises from the interaction of an agent with its environment. This issue is another aspect of the *frame of reference problem*, namely the *behavior-versus-mechanism* issue (e.g. addressed in [PS99, p. 112]).



(a) (b) (c) (d)

Figure 5.1: Subfigures (a) through (d) illustrate typical emerging behaviors in Experiment I. The snapshots are taken at times $t_a \approx 45000$, $t_b \approx 63000$, $t_c \approx 83000$ and $t_d \approx 83100$, respectively. Robots with six sensors are object-avoiders, robots with four sensors are driven by homeokinetic control. In Subfigure (a) the sensor's perceptive fields are included for one robot. The shade of the sensors indicate the intensity of their perception. A trajectory in world-coordinates is also shown for the past 200 units of time for all objects.

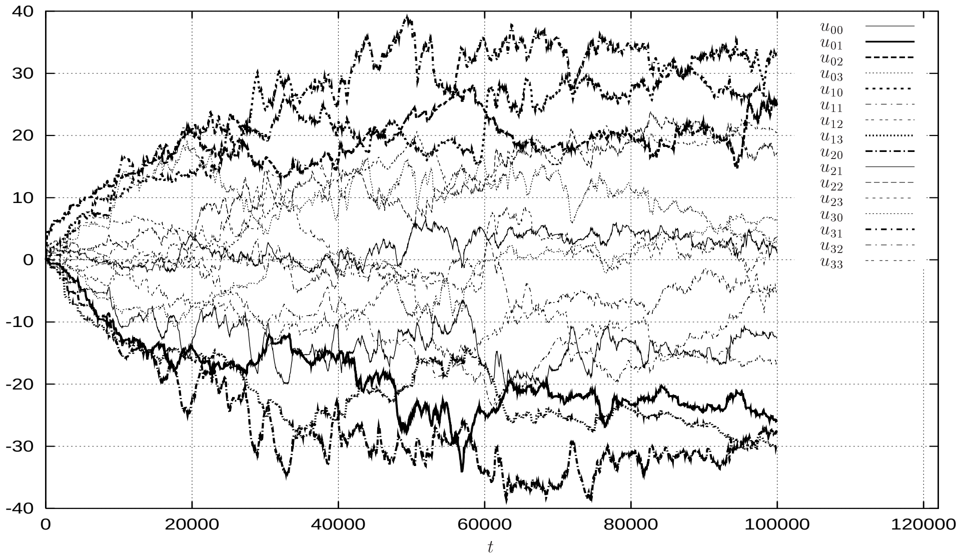


Figure 5.2: Evolution of the self-model's parameters u_{ij} over time for one robot in Experiment I (other robot's parameters evolve very similar). Since the illustration is somewhat cluttered, all parameters corresponding to adjacent sensors are shown with bold lines. They turn out to have the highest absolute values at the end of the experiment.

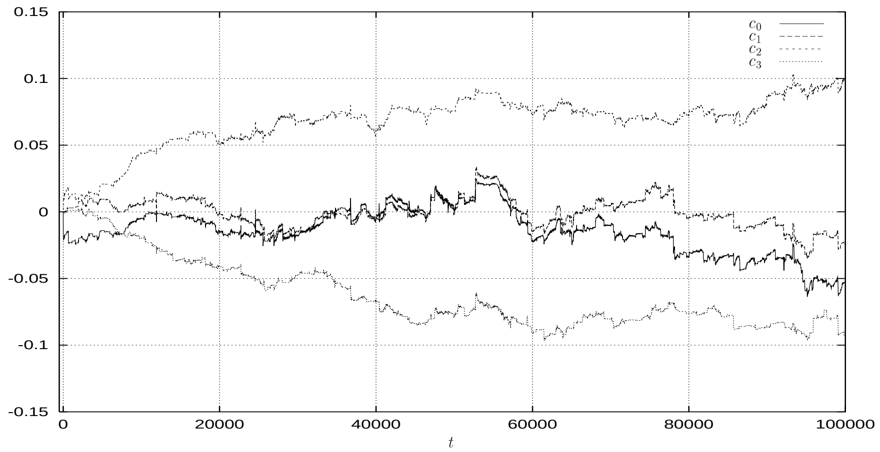


Figure 5.3: Evolution of the controller's parameters c_i in Experiment I for the same robot as in Figure 5.2.

5.5 Experiment IIa

In the next experiment a completely different world is considered, namely a world which is largely occupied by static objects forming a complex maze. The complexity for a navigating homeokinetic robot within this maze primarily comes from the hard predictability of sensor-changes resulting from its navigation since objects suddenly end or sharp corners might show up within sensor-range. A single homeokinetic robot within the maze was considered first.

Motivated by results obtained in [DP99] it is of course hoped for the robot to be able to develop some kind of obstacle avoidance keeping it from colliding with the static objects.
However, it turned out that emergence of this kind of behavior can not be obtained as easily as behaviors in the first experiment, specifically two main problems arise:

- While learning of the behaviors in Experiment I was completely self-reliable, this is not the case here: If no precautions are taken, the robot will just "learn" to approach any (the closest) wall either stay there or sliding along until a corner is reached, "satisfying" all its sensors with high perception-values and almost perfect predictability.
  A remedy applied, keeping the robot from adopting this kind of behavior, is to remove it from the wall anytime a collision occurs. While doing so, the sensor-values are shortly (for one controllerstep) set to 0 and thereafter take on the values depending on wherever the robot's new position is.
  Of course, this remedy is not completely satisfying since the statement, that homeokinesis provided a self-reliable framework, differing from related strategies such as evolutionary computation and reinforcement-learning in that it did not require externally defined fitness-functions or reward-distributions, has to be withdrawn in this context. This is because intervention, such as repositioning the robot, may be viewed as just another kind of formulation of an *objective-function* and actually proves to be similarly sensible as for the other strategies mentioned.

- The kind of behavior learned by the robot greatly depends on its initial position (even more so with every removal from a wall when a new position has to be chosen also).
  It was observed, that sometimes the robot (temporarily) developed quite an impressive maze-navigation capability, while at other times (with an altered initial position) it kept crashing into walls or developed a spinning motion. This phenomenon was observed even for very small

learning parameters (which have to be lowered quite a bit as compared to Experiment I in any way, since rapid fluctuations in sensor-values due to crashes would otherwise cause large steps in the parameter's evolution). One remedy was to introduce another (perfectly maze-navigating) robot into the world (with a fixed controller) which, on one hand-side, may seem as another trouble to be dealt with, but on the other side prevents any spinning-type of movement to remain stable, since sooner or later this companion will show up in the narrow corridors of the maze almost definitely resulting in a crash.

In Figure 5.4, the (partially) successful experiment is illustrated. The companion robot can be distinguished by noting only six sensors on it whilst the homeokinetic robot carries eight sensors.

5.5.1 Observations

In the beginning of the experiment a lot of crashes occur, whereupon the robot is placed in one of two positions depending on where its companion is located (it is placed in the position approximately furthest away from it wrt. its path within the maze), these positions are depicted in Subfigure (b). Upon repositioning the orientation is chosen such that it favors navigation with the same orientation as the companion.

After about 10.000 time-steps, the robot has learned almost collision-free maze-navigation and continues to circle the maze for about 10 times, only colliding once due to an encounter with the companion. After about 35.000 time-steps it spontaneously collides with a wall and completely "unlearns" all of its navigation-capabilities and seems to never recover.
Even for settings where no companion was placed in the world, no experiment could be conducted where (after acquisition of maze-navigation similar to the presented experiment) navigation remained stable.

5.5.2 Evolution of the Parameters

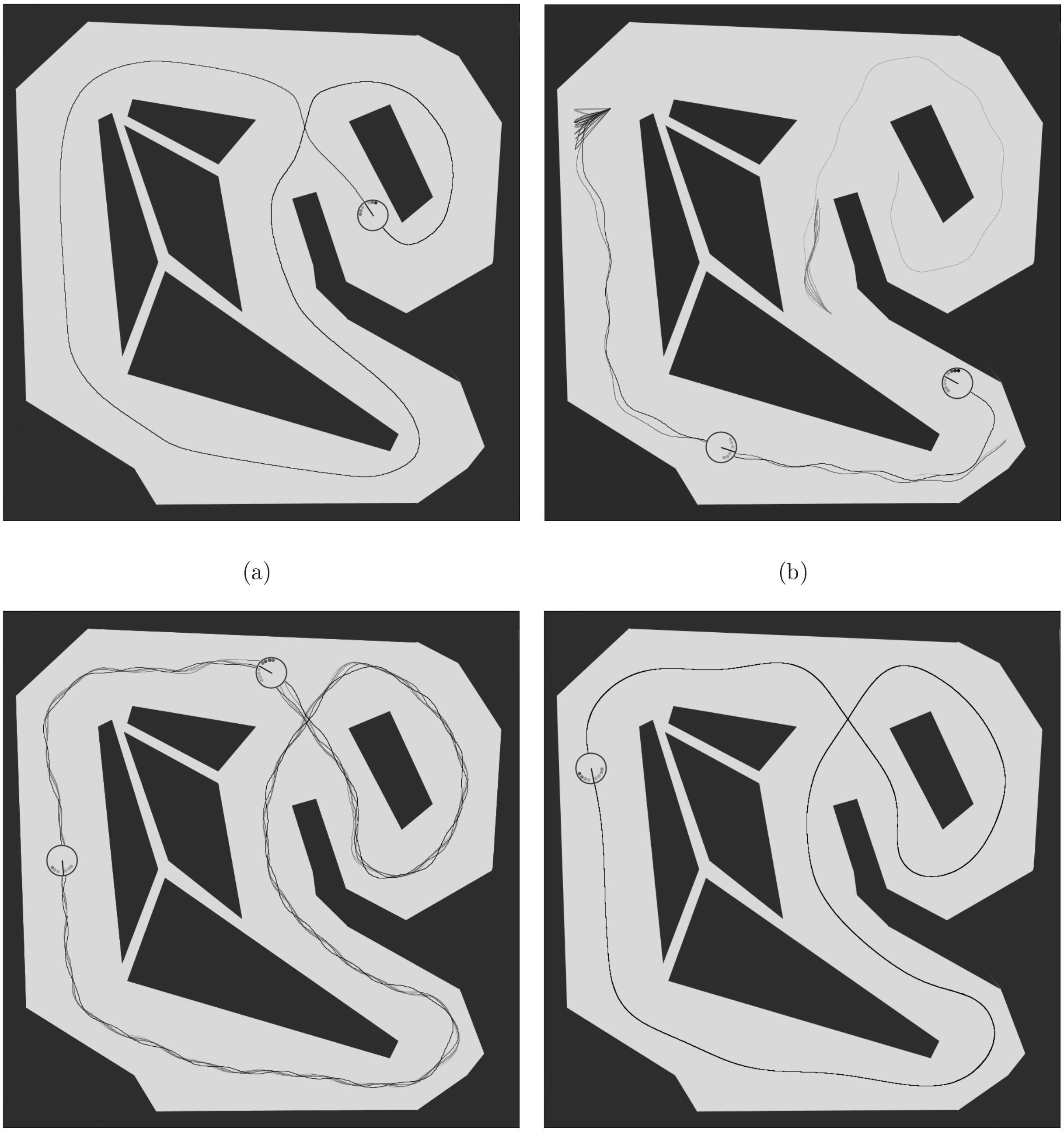
The evolution of parameters u_{ij} and c_i are given in Figure 5.5 and 5.6, respectively.

For times between 10.000 and 35.000 (approximately), evolution seems quite smooth with the exception of times 20.000 till 22.000 where a collision with the companion occurred shortly after step 20.000.
After time 35.000 the evolution takes on a very rugged form again and never recovers from it (which has been verified up to time 100.000, but only times smaller than 38.000 are shown in the figures).

5.5.3 Conclusion

On the one hand-side the experiment presented may appear hard since it really only allows for *perfect maze-navigation* to be called a meaningful behavior in this world (as opposed to Experiment I, where probably more reasonable behaviors are possible), on the other side the perfect companion-navigator operates with the same type of a controller and was initialized by hand quite easily, even functioning properly with only four of its sensors, leaving it with half of the sensors compared to the homeokinetic robot. It is therefore argued, that enough redundancy is provided to leave room for this kind of desirable behavior to emerge (and which will, as a matter of fact, prove true in Experiment IIb).

It is assumed at this point that homeokinetic control fails because of an insufficient self-model, since large fluctuations in the parameters imply large prediction-errors of the change in sensor-values.
A remedy for this problem will be presented in the next chapter, applying neural-networks to homekinetic control.



(a) (b) (c) (d)

Figure 5.4: Subfigure (a) shows the world for Experiment IIa including the perfectly navigating companion. In (b) the learning progress can be observed: Quite a few collisions have occurred already, but learning seems to be pretty promising. Subfigure (c) shows the homeokinetic robot going into its fourth collision-free round. In (d), the homeokinetic robot's controller was fixed (after two collision-free rounds), the periodic signal is removed from it and the companion is removed from the world, resulting in a very decisive trajectory.

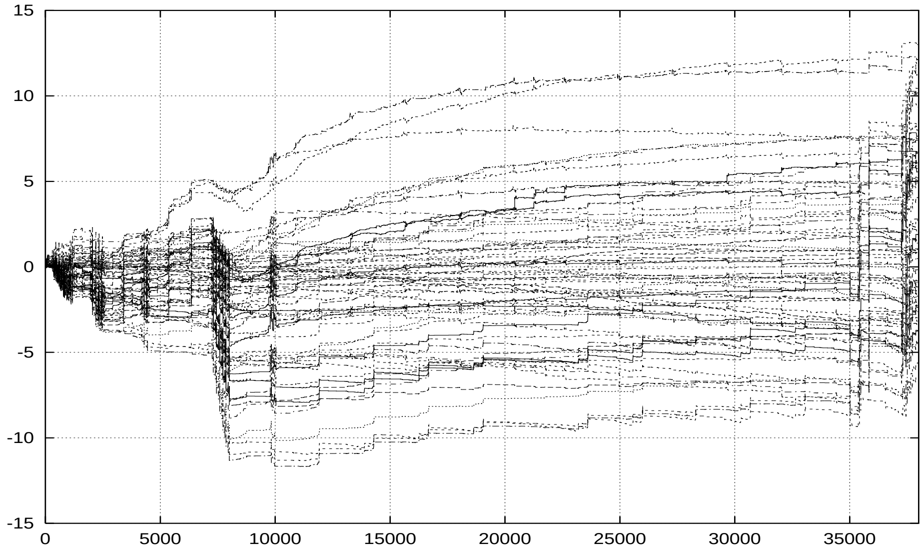


Figure 5.5: Evolution of the self-model's parameters u_{ij} over time. The parameters evolution is much smoother for times 10.000 until 35.000 (approximately), when the robot's navigation within the maze was very good.

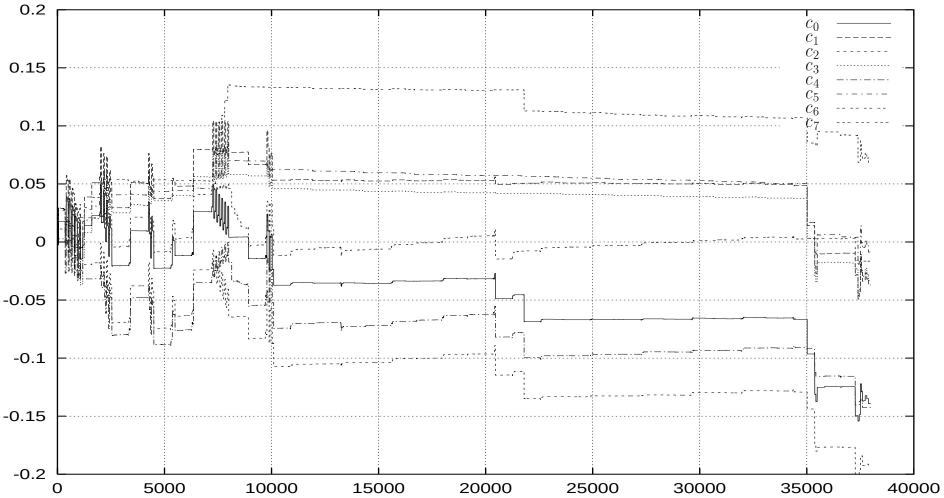


Figure 5.6: Evolution of the controller's parameters c_i. Their values seems to be very stabilized for the time-interval in which maze navigation was very good, with the exception of times between 20.000 and 22.000 (approximately), after the accompanied robot was encountered.

Chapter 6

Application of Recurrent Neural Networks to Homeokinesis

Many loves in a lifetime, they seem to go astray,
many people come and go, they never seem to stay,
sometimes I'm kiddin' myself, then I realize,
I'm just living in an artificial paradise.

Plastic state of mine, superficial blow,
livin' in a vacuum, close to zero,
sometime I kid myself, then I realize,
I'm just living in an artificial paradise.

[...]

— J. J. Cale

It was concluded in the last chapter from Experiment IIa that homeokinetic control in its presented form is capable of providing a basis for emergence of *reasonable* behavior in a quite complex world (i.e. a maze), but might still fail in stabilizing this behavior.

Since the controller evolved quite favorably (i.e. was capable of reliable maze-navigation) for quite a while, it is suspected of the problem to be rooted in the nature of the self-model rather than the controller.
Specifically, the realization as a bilinear self-model given by equation(5.3.6) is argued to be not suited for the kind of world in Experiment II, while it showed to be suited very well for Experiment I (a meaningful interpretation of the evolved parameters could even be found).
As a consequence of this argument, another formalization for the self-model in homeokinetic control is considered in this chapter, while the controller remains as it was introduced in the last chapter.

6.1 Recurrent Neural Networks

The proposed self-model will have the structure of a *recurrent neural network* (RNN) where current sensor-values and controller-output serve as the input and the desired output is the change in sensor-values at the next controller-step.

6.1.1 Architecture

The neural network architecture is depicted in Figure 6.1, where each neuron receives all external inputs $z_1(t) \dots z_m(t)$ as well as all internal inputs $z_{m+1}(t) \dots z_{m+n}(t)$, receiving the outputs of all neurons at the previous time-step. Altogether there are m external inputs and n internal inputs plus another input $z_0(t) = 1$ (not shown in the figure), which serves as a pseudo-input to introduce a bias for each neuron.

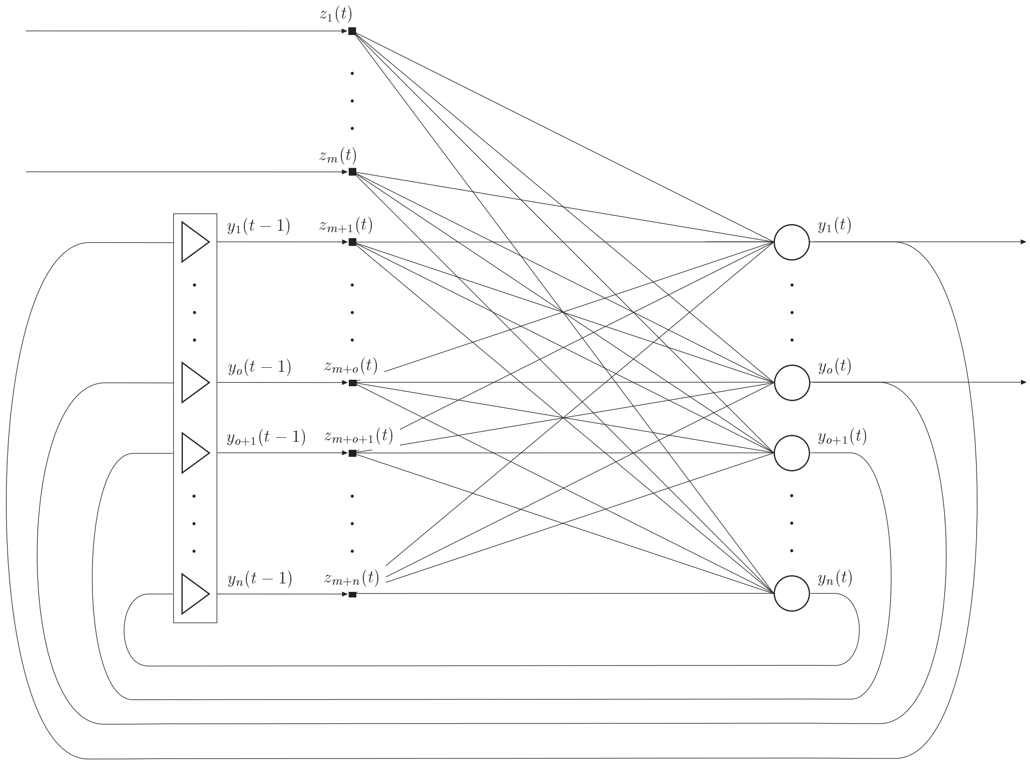


Figure 6.1: Illustration of the architecture of a general recurrent neural network. Circles depict neurons, the small square boxes symbolize the input values to the network which are weighted and the array of triangles depict 1-step-delay-elements.

Neurons 1 through o are output-neurons (i.e. their output is available externally), whereas neurons $o+1$ through n are hidden-neurons, meaning their output is not visible externally but is only fed back into the network. The neuron j's input z_i is hereby weighed by an adaptive weight w_{ij}. A RNN's set of all parameters is thus given by a $n \times (m+n+1)$ matrix W. Neuron j's output $y_j(t)$ at time t is given by the expression:

$$y_j(t) = \varphi\left(\sum_i w_{ij}\, z_i(t)\right), \tag{6.1.1}$$

where φ denotes the neuron's activation-function.

6.1.2 Gradient Learning in RNNs

Quite a few gradient-learning algorithms have been developed for RNNs operating in discrete-time like *backpropagation-through-time*, *recurrent backpropagation*, *real-time-recurrent-learning* and other learning algorithms (e.g. summarized in [HKP]), of which some are restricted to less general architectures than the one considered here.

The learning-algorithm chosen here is real-time-recurrent learning (RTRL) since it allows for on-line learning, that is learning while a (time-) sequence of data is presented rather than after it is completed which is just what is in need for an adaptive self-model in the general setting of homeokinesis.
Another reason for choosing RTRL is that it's memory-requirements do not depend on the length of the training-sequence as is the case in backpropagation-through-time, for example ([WZ95]).

The algorithm to be presented was originally proposed in [WZ89] and its derivation presented here follows the lines in this article with the exception that the network is defined to produce an output at time t based on a corresponding input at time t (as opposed to a previous input at time $t-1$) in order to preserve and emphasize the interpretation, in faith of the last chapter, that the self-model is to *predict*, based on current action, what will be experienced in the future.
For gradient-descent to be applicable, an error measure for the total accumulated error over a time-interval $[t_0, t]$ is introduced:

$$E_{total}(t_0, t) = \sum_{t=t_0}^{t} E(t), \tag{6.1.2}$$

where $E(t)$ is the instantaneous sum of squared errors of all output-neurons at time t:

$$E(t) = \frac{1}{2} \sum_{j=1}^{o} (e_j(t))^2 , \tag{6.1.3}$$

where $e_j(t) = d_j(t) - y_j(t)$ denotes the error of neuron j at time t, namely the difference between desired output and actual output.

The main idea of RTRL is now to perform *online* gradient descent along $\frac{\partial}{\partial W} E(t_0, t)$, such that instead of updating weight w_{ij} once for the whole time-interval $[t_0, t]$ (often called an epoch) it is updated at each time-step according to the gradient of the instantaneous error $E(t)$:

$$\triangle w_{ij}(t) = -\eta \frac{\partial}{\partial w_{ij}} E(t), \tag{6.1.4}$$

where η is the learning rate parameter.

Of course, online training as described does not follow the exact gradient of the total error, but according to [Hay99] this inaccuracy has shown to be negligible in practice if η is chosen small enough and it relates to *backpropagation-through-time* (following the total error) just as *online backpropagation* relates to *batch-mode-backpropagation* in feedworward neural networks.
The gradient of the instantaneous error $E(t)$ is given by:

$$\frac{\partial}{\partial w_{ij}} E(t) = - \sum_{k=1}^{o} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}. \tag{6.1.5}$$

Introducing a shorthand-notation v_k for the weighted sum of inputs to neuron k, the partial derivative of neuron k's output in equation (6.1.5) may be expressed as:

$$\frac{\partial y_k(t)}{\partial w_{ij}} = \varphi'(v_k(t)) \left(\delta_{ik}\, z_j(t) + \sum_{l=o+1}^{n} w_{kl} \frac{\partial y_l(t-1)}{\partial w_{ij}} \right), \tag{6.1.6}$$

where δ denotes the Kronecker-delta symbol.
Introducing another shorthand-notation λ_{ij}^k for the partial derivative of y_k wrt. w_{ij} and writing the above equation as a discrete-time dynamical system yields:

$$\lambda_{ij}^k(t) = \varphi'(v_k(t)) \left(\delta_{ik}\, z_j(t) + \sum_{l=o+1}^{n} w_{kl}\, \lambda_{ij}^l(t-1) \right). \tag{6.1.7}$$

Reasonable initial conditions are given by:

$$\lambda_{ij}^{k}(0) = 0 \ \ (\forall i, j, k). \tag{6.1.8}$$

By updating all λ_{ij}^{k} values in this discrete-time dynamical system at each time-step, the update for weight w_{ij} is thus given by:

$$\triangle w_{ij}(t) = \eta \sum_{k} e_k(t)\, \lambda_{ij}^{k}(t), \tag{6.1.9}$$

completing the derivation of RTRL (online gradient-) learning.

6.1.3 Teacher-Forcing

Teacher-forcing is a technique often applied to (supervised) learning in recurrent neural networks. It states that the actual output of any neuron is to be replaced by its desired counterpart in subsequent computations, where that output is fed back into the network. It is thus only applicable to output-neurons (since no desired output is defined for hidden-neurons).

According to [Hay99], teacher forcing may provide at least the following beneficial effects during online-learning:

- It may lead to faster training. (Since it amounts to the assumption that the network has provided correct output in the previous learning-steps).
- It may serve as a corrective mechanism during training. Considering, for example, a network with optimal weights but on accident currently operating in the wrong region of state-space, teacher forcing would immediately push the network back into the right spot in state-space thus avoiding any harmful weight-adjustments.

Despite the fact that in addition to approximation of the total (epochwise-) error in RTRL, teacher-forcing effectively results in yet another modification of pure gradient-descent wrt. the total error, it has shown to work very well in the experiments conducted (i.e. no instance has been found where a regularly trained RNN-self-model performed better than its teacher-forced counterpart).

6.1.4 A RNN-Self-Model - Practical Considerations

The application of recurrent neural networks to homeokinetic self-models is straight-forward. The sensor-values plus the controller's output serve as the input to the network on each controller-step and the network's architecture has to be set up such that the number of output-neurons match the numbers of the robot's sensors.

In the conducted experiments the function $\varphi(x) = k\tanh(x)$ is always chosen as the neuron's activation-function, where k is a constant determining the absolute maximum possible prediction of change in sensor-values between successive controller-steps of the RNN. Typically k was obtained by sampling a few sensor-values for controllers producing random movement and selecting the highest recorded change.
Similarly, the input ω was always scaled such, that its maximum absolute value ω_{max} was approximately of the same order of magnitude as the sensor-values, such that the corresponding synaptic weights (ideally) learn at approximately the same speed as the weights of the other (external) inputs.

Learning was modified by including a momentum-term in the update equations, where it was ensured that the weight-update at any given time-step is never greater (in absolute value) than some predefined small value to avoid "large" weight-adjustments.
The initial synaptic weights in the network were set to small values according to a uniform distribution with mean 0.

6.2 Experiment IIb

In this section, Experiment IIa from the last chapter will be revisited with a RNN functioning as a self-model in the homeokinetic framework.
As the observations are reported they are compared with those of Experiment IIa. The original self-model will be referred to as the bilinear self-model and the revised version will be referred of as the RNN-self-model.
Different RNN-architectures have been tested prior to Experiment IIb and an architecture with no hidden neurons proved to be completely sufficient for the prediction task and was thus favored over any more complex architecture.

6.2.1 Observations

As was the case in Experiment IIa, learning still proved to be very sensitive to initial conditions, learning-rate and other parameters.
The only changes made for the revisited experiment concern different initial positions of the robot (including the cases, when repositioning was necessary after either one of the robots crashed into a static object), the learning-rate parameter which was changed from 0.15 to 0.85 and the noise's period, which was reduced to about 2/3 as compared to Experiment IIa. (For the original settings, no successful experiment could be conducted without changing too

many other parameters - the same holds vice versa, the setting used in part b of the experiment resulted in spinning behavior for part a).

In all cases where maze-navigation behavior emerged, this behavior proved to be stable with the RNN-self-model (at least until time 50.000 which usually was the maximum simulation-time). For the specific setting presented in Experiment IIb, this was verified for times up till 100.000.
Three snapshots of the experiment are reported in Figure 6.2: While maze-navigation is not yet acquired at its most advanced level in Subfigure (a) and while it is perfect in Subfigure (b). The trajectory through the maze is very similar to Experiment II with the notable difference that (with the added controller-noise) it is more decisive, i.e. successive navigation cycles through the maze follow each others trajectories much closer. However, it is not clear whether this effect is due to superior adjustment of the controller's parameters or whether it is caused by the reduced period of the noise-signal. In Subfigure (c), maze navigation is shown for the matured robot with periodic noise removed from its controller-output and the companion-robot removed from the world.

The learning-process was accelerated compared to learning with the bilinear self-model. The first 5000 elapsed time-steps without any crashes occurring could be reported at (approximately) time 11.500 (compared to 15.100 for the previous experiment). As illustrated in Figure 6.3, the mean-squared error resulting from RNN-predictions (where the mean was taken over a time-window of length 6000) was smaller by a factor of between five and ten at times where maze-navigation behavior was acquired and still much smaller on other times (hence the much higher learning-rate for the controller).
Thus, the prediction-error could be lowered *tremendously*.
However, it has to be kept in mind that these values may not be directly comparable especially because of the lowered noise's period. This issue will be addressed again in section 6.3 at the end of this chapter.

6.2.2 Evolution of the Parameters

The evolution of controller-parameters c_i is depicted in Figure 6.4. For times greater than approximately 6500, (almost) perfect maze-navigation behavior was acquired and remained stable. One spontaneous collision still occurred at around time 37.000, but the controller recovered shortly thereafter.

Evolution of the RNN's parameters (synaptic weights) can be witnessed in Figure 6.5, where most parameters seem to evolve rather smoothly except for some which are oscillating around 0.

Since not much more information can be extracted from the cluttered figure for all RNN-parameters (there are 144 of them), two distinct sets are extracted and plotted separately in Figure 6.6 and 6.7.

The first set contains all those weights which belong to *self-recurrent* connections (connecting the past output of a neuron to one of its inputs). It is witnessed here, that all such connection-weights are increased throughout the learning process and seem to be attracted to positive values (which was verified by running the same example for a longer time-interval). This observation may be interpreted as a justification of a *recurrent* architecture for the self-model since obviously recurrence is useful for predictions (otherwise the corresponding feedback-weights would not be increasing).

The second set of weights depicted separately in Figure 6.7 contains all weights connecting the input corresponding to angular-velocity with any neuron. Here, an interesting effect can be observed which is in some way related to Figure 5.2 on page 66, namely angular velocity causes predictions for sensors on the side of the robot where it is turning to (depending on the sign of ω) to be increased. Although still oscillating quite a bit, the weights seem to stabilize at a level for times greater than 20.000.

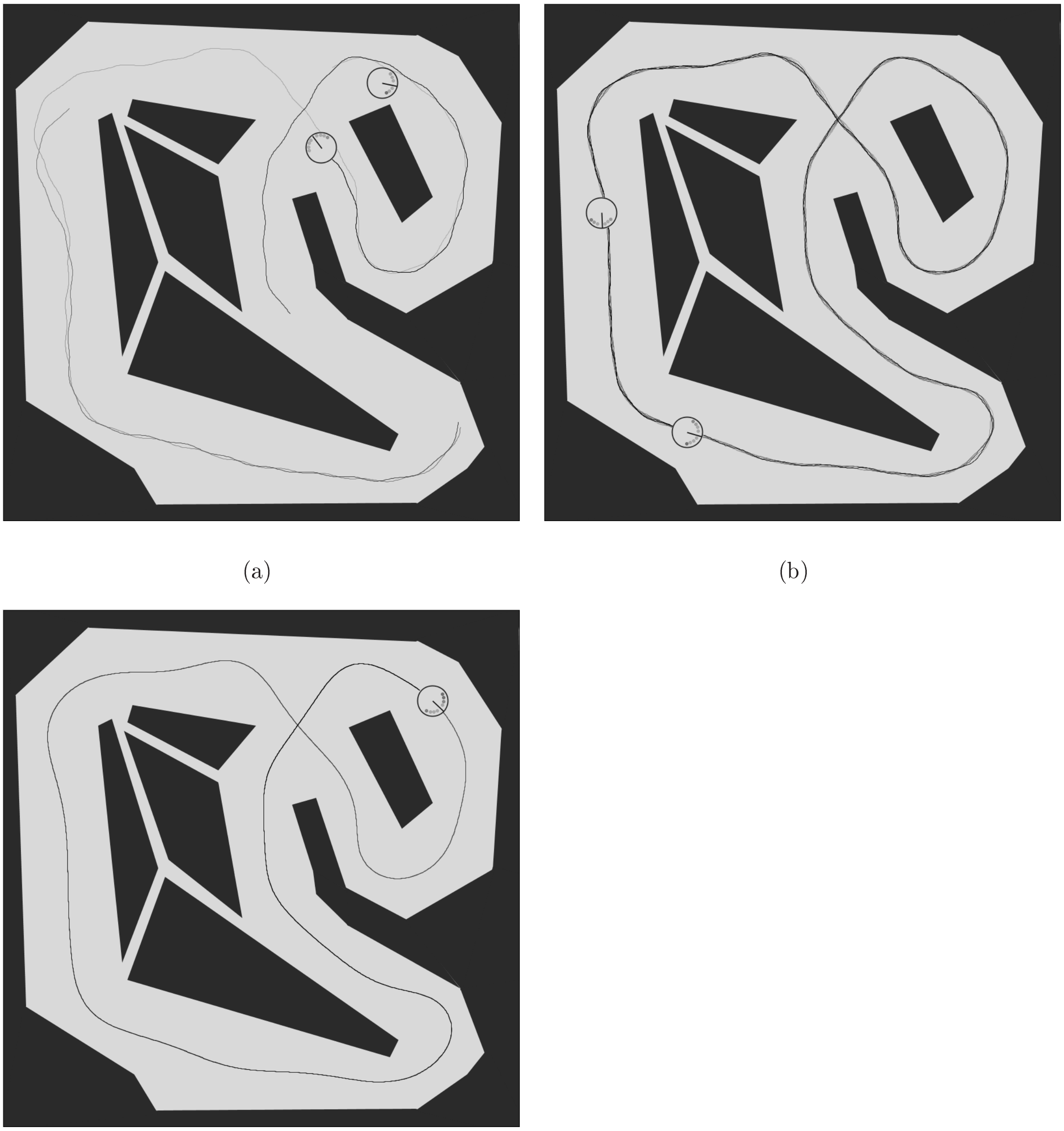
6.2.3 Adopting the Robot's Perspective

As a final presentation of collected data and its interpretation, the robot's trajectory in sensor-space is depicted in Figure 6.8 and 6.9, where 8-dimensional sensor space has been projected into three dimensions for two different combinations of three sensors.
Both trajectories display data from the same time-series, which was recorded for a robot with fixed controller after having acquired perfect maze-navigation skills (precisely after the first 5000 collision-free time-steps) and where the noise signal is removed.
In the figures, the trajectory has been plotted with lines for exactly one round in the maze and with dots for the remaining rounds (approximately 40). It may thus be witnessed, that the trajectory in sensor-space is very decisive with only a few values breaking away from it.

Additionally it is quite comprehensible from the first figure, how the robot navigates through the maze: When no object is immediately in front of the robot, the values $s_0(t)$ and $s_1(t)$, corresponding to the two frontal sensors, are low. Whenever an object appears, the robot passes by, leaving it on one of its sides. This is reflected by the fact that the trajectory engages in cycles with either high values of $s_0(t)$ or $s_1(t)$, but regions with both values being high are completely left out by the trajectory.

6.2.4 Conclusion

In Experiment IIb, RNNs have proved to be well applicable to the home-okinetic framework and seem to provide quite an advance over the original (bilinear) self-model at least in the setting considered.
Since some residual doubt may remain, that comparison of parts a and b of Experiment II may be misleading because of the changed settings (initial positions, learning-rate and especially the noise's period), another small deviation of Experiment II is presented in section 6.3, leading to further (stronger) evidence that a RNN as a self-model is superior to a bilinear model in this setting.



(a)

(b)

(c)

Figure 6.2: Subfigure (a) shows the learning progress at a mature stage, the only remaining hotspot in collision-free navigation seems to be in the lower right corner. In (b) the robot has learned perfect navigation in the maze. Subfigure (c) illustrates the robot's path with fixed controller (after the first two collision-free rounds) and removed periodic signal in a world where its companion is removed.

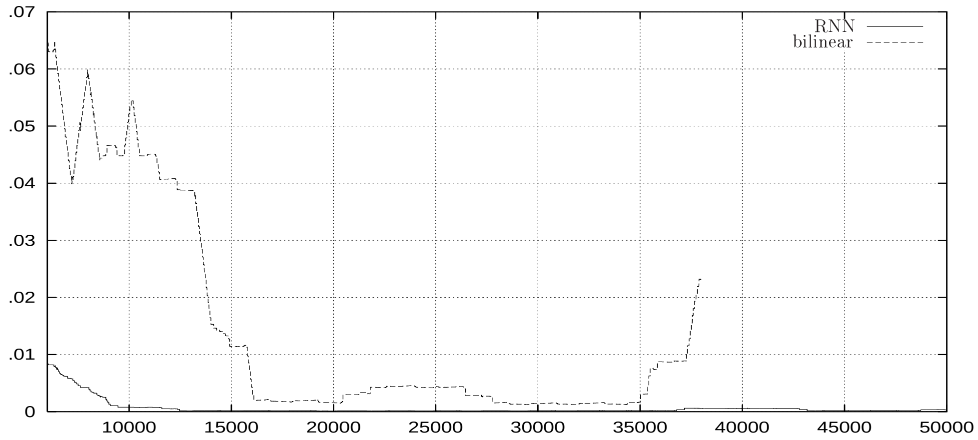


Figure 6.3: Illustration of the mean squared error in experiments IIa and IIb. RNN prediction seems to be quite an improvement over the bilinear model.

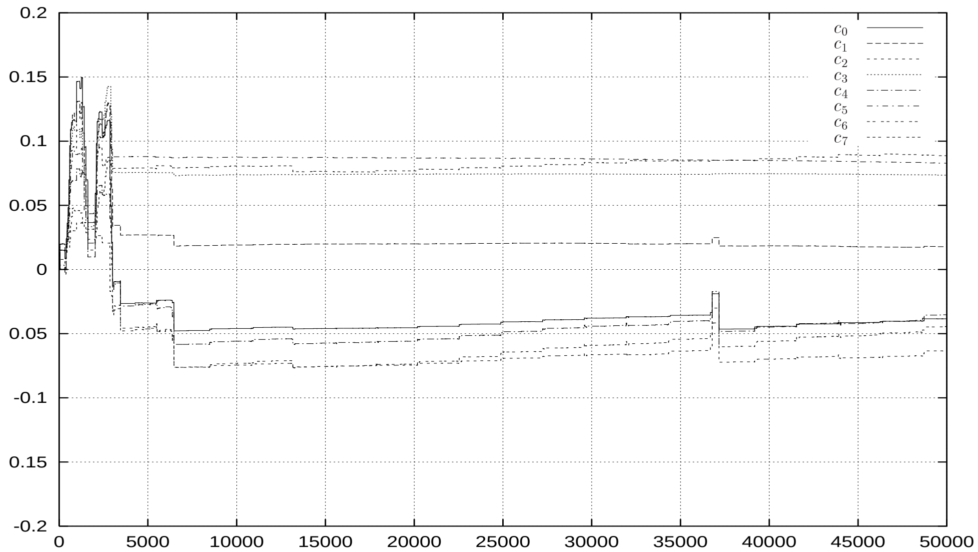


Figure 6.4: Illustration of the evolution of controller-parameters c_i. Perfect maze-navigation was learned until time-step 6500, approximately.

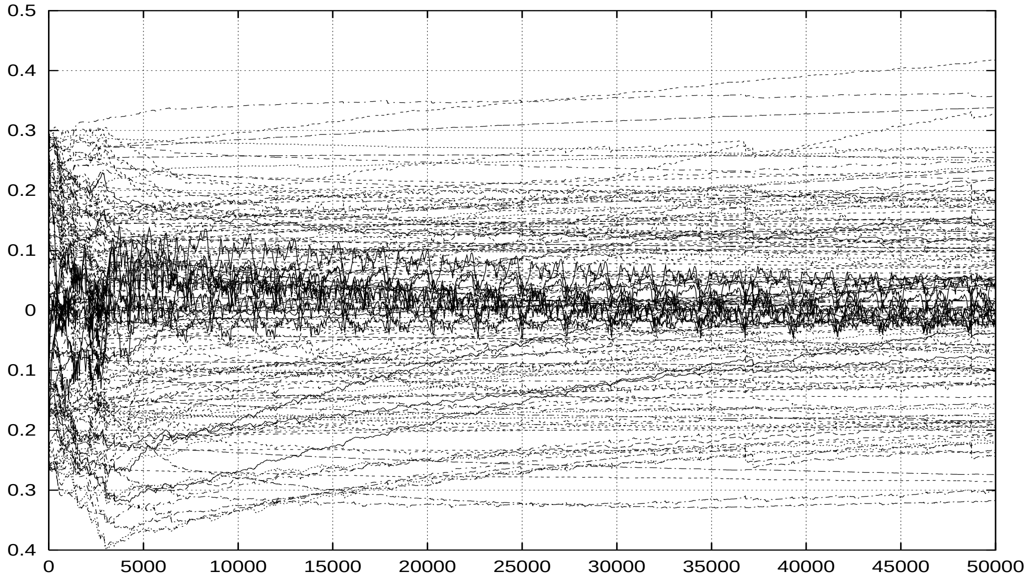


Figure 6.5: Illustration of the evolution over time of parameters w_{ij} (synaptic weights) in the RNN. Evolution appears to be smooth, once good maze-navigation skills have emerged. The total number of weights is 144.

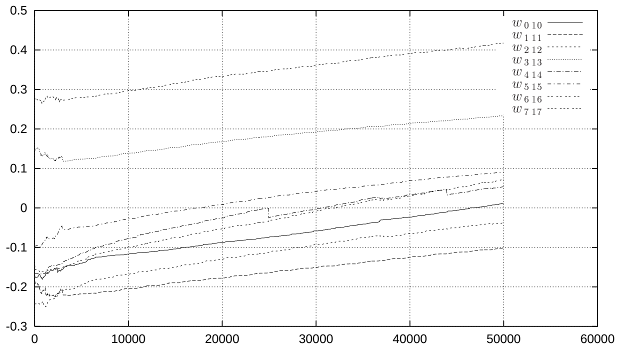


Figure 6.6: All *self-recurrent* weights are extracted from Figure 6.5, here. As learning proceeds, they increase and tend towards positive values.

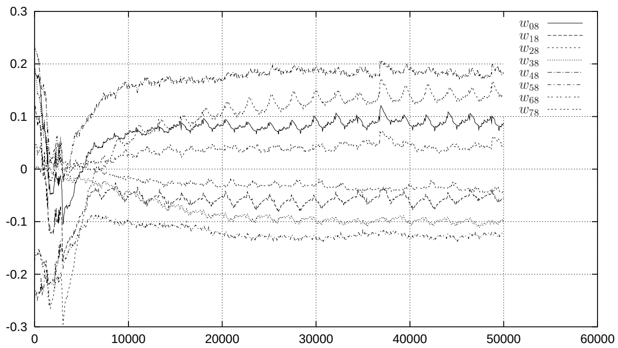


Figure 6.7: Illustrated are all weights originating at the input for angular velocity ω. All those weights connecting to sensors located on the left of the robot's "nose" are positive and others are negative.

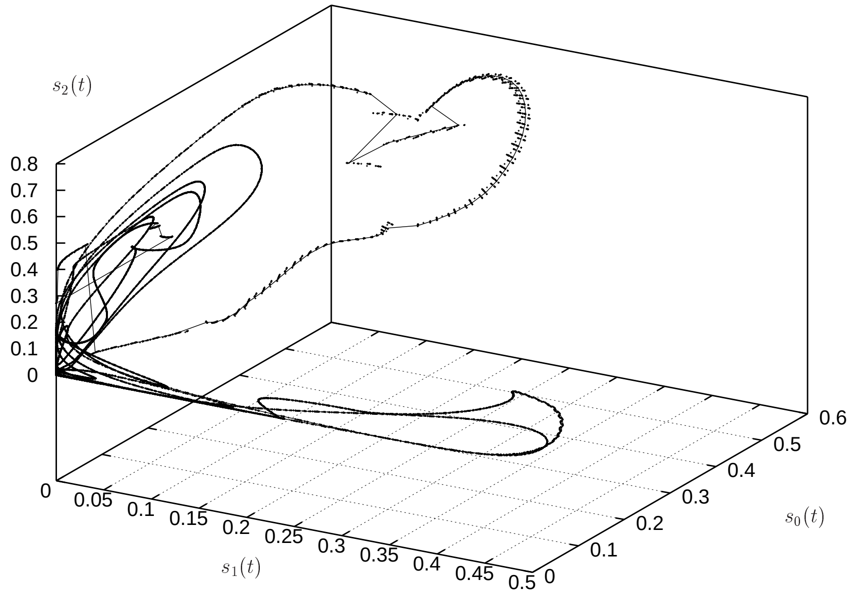


Figure 6.8: Illustration of the trajectory in sensor-space. Depicted is a projection of the 8-dimensional trajectory into three dimensions, corresponding to three sensors in the front of the robot. Data is collected for 100.000 time-steps corresponding to approximately 40 maze-rounds of the robot with fixed controller after maze-navigation skills have been acquired and with removed periodic noise-signal.

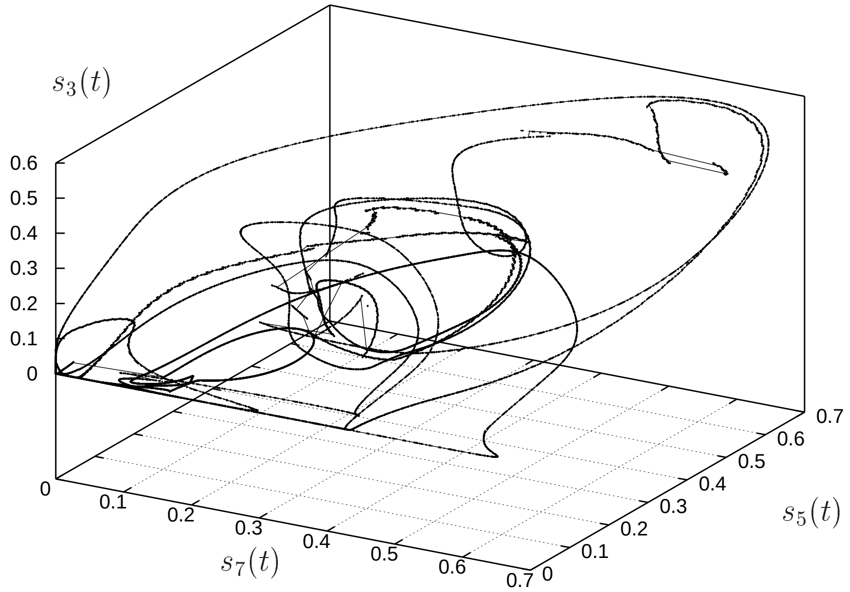


Figure 6.9: The same data as in 6.8 serves as a basis, but projection is into three dimensions corresponding to three sensors on the right wrt. the robot's nose.

6.3 Experiment IIc

Finally, another Experiment (IIc) is conducted, where basically Experiment IIa is rerun another time, but with the addition that besides the bilinear self-model, a RNN is trained concurrently for the same robot (that is to say for the same controller which is still trained according to the prediction error of the bilinear self-model).
Otherwise no changes are made to the exact original setting.

6.3.1 RNN-Self-Model vs. Bilinear Self-Model

This setting is not argued to provide a "fair" basis for comparison, but (if it does not provide a "fair" basis) favors the *bilinear* model, since this is the one regulating the controller's adaption and thus the controller (ideally) tends to compensate *its* errors.
Only the mean-squared error for both models are presented (in Figure 6.10 and Figure 6.11), providing very strong evidence, that a RNN-self-model is indeed superior to its bilinear counterpart.

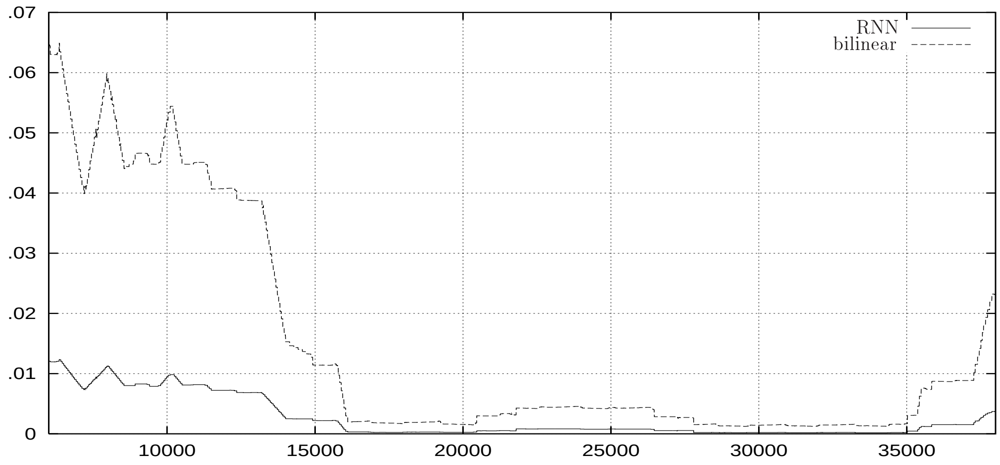


Figure 6.10: Illustration of the mean squared error in experiment IIc. RNN prediction clearly outperforms the bilinear model.

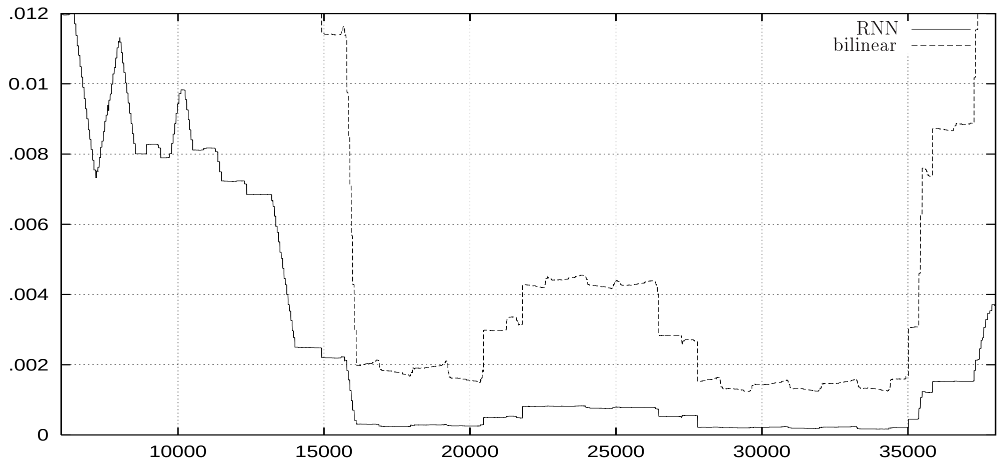


Figure 6.11: Magnification of Figure 6.10 for times, when maze-navigation was very good.

Chapter 7

Conclusion

This thesis was concerned with modeling of autonomous robots, in order to simulate and investigate emergent agent-behavior embodied in a dynamic environment, which are controlled by the principle of homeokinesis.

In Chapters 2 and 3 appropriate models of the physical entities (including the robots) have been developed. In Chapter 4, these models were embedded in the discrete-event-specification framework in order to simulate them.
In Chapters 5 and 6 both, the models as well as the simulator, were utilized for experiments to investigate the principle of homeokinesis and proved to be very well suited for this task.
Investigation of the homeokinetic framework resulted in emergence of a variety of behaviors in the experiments. The obtained results were presented in detail and conclusions have been drawn for all experiments.

In one experiment (IIa), presented in Chapter 5, the original homeokinetic framework resulted in emergence of a reasonable behavior, but failed in stabilizing it. As a consequence, remedy was proposed and developed in Chapter 6 by utilizing recurrent neural-networks.
In Experiment IIb, the proposed remedy succeeded in stabilization of the same behavior that the original framework had failed to stabilize. It was thus conjectured that application of recurrent neural-networks may improve the original homeokinetic framework, which was verified by Experiment IIc.

As a final conclusion to the investigations and conducted experiments it is therefore conjectured that (in relation to the original homeokinetic framework), application of recurrent neural-networks may lead to more *well-guided* learning due to improved predictions of (perceived) effects of actions in the environment, such that learning is more focused on avoiding those (fewer) actions and/or situations with unpredictable effects.

Bibliography

[Arv91] James Arvo, editor. *Graphic Gems*, number II, Boston, 1991. Academic Press.

[Cal92] J. J. Cale. Artificial paradise. *Number 10, track 5.* Silvertone Records, 1992.

[Der00] Ralf Der. Self-organized robot behavior from the principle of homeokinesis. In *Proceedings of the SOAVE Workshop*, Ilmenau, Germany, May 2000.

[Dil99] Antoni Diller. *LaTeX Line by Line.* John Wiley & Sons, West Sussex, England, second edition, 1999.

[DP99] Ralf Der and Thomas Pantzer. Emergent robot behavior from the principle of homeokinesis. In A. Löffler, F. Mondada, and U. Rückert, editors, *Proceedings of the* 1^{st} *International Khepera Workshop*, Paderborn, Germany, December 1999. Heinz-Nixdorf-Institut Verlagsschriftenreihe.

[DSP99] R. Der, U. Steinmetz, and F. Pasemann. Homeokinesis - a new principle to back up evolution with learning. In M. Mohammadian, editor, *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA'99)*, Vienna, February 1999. IOS Press.

[Gla90] Anrew S. Glassner, editor. *Graphic Gems*, Boston, 1990. Academic Press.

[Hay99] Simon Haykin. *Neural Networks: A comprehensive Foundation.* Prentice-Hall, Upper Saddle River, New Jersey, second edition, 1999.

[HKl89] W. Heyroth, W. Kuhn, and H. Glaßl. *Mechanik.* VEB Deutscher Verlag der Wissenschaften, Berlin, 1989.

[HKP] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction To The Theory of Neural Computation - A Lecture Notes Volume in the Studies in the Sciences of Complexity*. Addison-Wesley.

[Kli96] J. Frederick Klingener. Programming combined discrete-continuous simulation models for performance. In J. M. Charnes, D. M. Morrice, D. T. Brunner, and J. J. Swaine, editors, *Proceedings of the 1996 Winter Simulation Conference*, Coronado, California, December 1996.

[PS99] Rolf Pfeifer and Christian Scheier. *Understanding Intelligence*. MIT Press, Cambridge, Massachusetts, 1999.

[Sto93] Josef Stoer. *Numerische Mathematik 1*. Springer Verlag, Berlin, Heidelberg, 6. korrigierte Neuauflage, 1993.

[Woa00] Graham Woan. *The Cambridge Handbook of Physics Formulas*. Cambridge University Press, Cambridge, UK, 2000.

[WZ89] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, Volume I:270–280, 1989.

[WZ95] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, pages 433–486. Lawrence Erlbaum Publishers, Hillsdale, N.J., 1995.

[Zei76] Bernard P. Zeigler. *Theory of Modeling and Simulation*. John Wiley & Sons, New York, 1976.

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, San Diego, second edition, 2000.

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 01.04.2001 (Robert Pies)