D

DEUTSCHES ELEKTRONEN-SYNCHROTRON

DESY-THESIS-2000-018
May 2000

Enzymatic Simulation of Complex Processes

by

J. Würthner

ISSN 1435-8085

NOTKESTRASSE 85 - 22607 HAMBURG

Enzymatic Simulation of Complex Processes

Dissertation
zur Erlangung des Doktorgrades
des Fachbereiches Physik
der Universität Hamburg

vorgelegt von
Jan Würthner
aus Hamburg

Hamburg 2000

Gutachter der Dissertation:	Prof. Dr. G. Mack
	Prof. Dr. K. Fredenhagen
Gutachter der Disputation:	Prof. Dr. G. Mack
	Prof. Dr. J. Bartels
Datum der Disputation:	14.4.2000
Sprecher des Fachbereichs Physik und Vorsitzender des Promotionsausschusses:	Prof. Dr. F.W. Büsser

Abstract

Computer simulations have become very important in todays sciences. The theory of complex systems (short: *systems theory*) provides an appropriate basis for abstract modeling sciences - in particular physics.
In this work a software package is introduced, which serves as a general modeling and simulation tool.
Beginning with a brief introduction to systems theory, modern techniques of software development are discussed. It is found that the object oriented approach and the theory of complex systems share many concepts. Making extensive use of object oriented techniques, a software has been developed, which consists of three components: The *core package*, responsible for modeling system theoretical scenarios, the *presentation package*, which provides a user interface, and a *language interpreter*, a powerful scripting tool acting as an interface between the core and the presentation.
As a general simulation tool, it may be imaginable to simulate processes of life. This work concludes with a short description of life processes in nature as well as a section about artificial life.

Zusammenfassung

Computersimmulationen haben in der heutigen Wissenschaft einen wichtigen Stellenwert. Die Theory komplexer Systeme (kurz *Systemtheorie*) bietet eine angemessene Basis für abstrakte Modellierung wissenschaftlischer - insbesondere physikalischer - Prozesse.
In dieser Arbeit wird ein Softwarepaket vorgestellt, das als allgemeines Simulationswerkzeug fungiert.
Nach einer kurzen Einführung in die Systemtheorie werden aktuelle Methoden der Softwareentwicklung erläutert. Es zeigt sich, dass der objektorientierte Ansatz konzeptionell kompatibel zur Systemtheorie ist. Unter massivem Gebrauch objektorientierter Methoden wurde ein Softwarepaket entwickelt, das sich in drei Komponenten gliedert; den *Kern* als Fundament zur Modellierung Systemtheoretischer Szenarios, der *Präsentation* zur Darstellung der Elemente der Kernkomponente sowie einer Benutzerschnittstelle, und einem *Interpreter*, der mittels einer leistungsfähigen Skripsprache eine Schnittstelle zwischen dem Kern und der Präsentation bildet.
Als ein allgemeines Simulationswerkzeug ist es denkbar damit auch Lebenprozesse zu simulieren. Eine knappe Beschreibung biologischer Lebenprozesse sowie ein Kapitel über künstliches Leben beschliessen diese Arbeit.

Contents

A Modern Version of Platos Concept of Ideas

A biologist and a system theorist meet in a cafeteria and argue about their disciplines. At some point the biologist utters thoughtfully: "*Well, one of the problems of biology is the lack of exactness. Physical quantities are well defined. If you want to determine a systems behaviour in time, you can use the quantum physical framework and calculate* exactly *the way it develops. Even for large systems one can use computers to cope with the large amount of information. - In biology we might consider a flying bird and ask when it will change its direction. But to calculate this, we needed a detailed physical description of a bird in terms of atoms and molecules, which we do not have.*"
So they discuss, what such a definition in terms of atoms and molecules might look like. But with any attempt to define a bird at the atomic scale, they could think of a counter example that would not fulfill the definition, but still be said to be a bird. Just like Platos *idea* of a table is not characterized by its material or shape, objects in general might as well be characterized by their interaction with the environment rather than by their internal structure.
They finally agree that such a microscopic definition might be useful in physics, but definitely misses the point in biology, and that an interdisciplinary language to formulate problems from different disciplines might lead to just the required level of exactness in each scale.

Chapter 1

INTRODUCTION

This chapter shall introduce the basic ideas of systems theory

The aim of scientific theories is to provide a model of nature. It is assumed that only few fundamental principles underly all natural phenomena. In order to increase generality, new theories have always been introduced along with a reduction of presupposed structure and thus reduction of a priory complexity. Einsteins principle of coordinate independence and the turning away from the classical particle viewpoint in quantum physics are good examples. *A theory will be the more fundamental the less structure is assumed* a priory. *This is plausible, because what is assumed is not explained. [13]*
The *Theory of Complex Systems (ToCS)* tackles the challenge to cope with a minimum of *a priori* structure. It intends to provide methods to describe arbitrary systems intelligible for the human intellect. This is not only a very sophisticated task, it also brings up the question: What and how does the human mind think?
Characteristic for the system theoretical point of view is the attempt to get to the idea of complex systems by ascribing structure not to the system rather than to the describing object, in other words: *structure arises in the eye of the beholder.* A basic assumption about the structure of human thinking can be formulated informally as a pre-axiom:

The human mind thinks about relations
between things or agents. [13]

Thus the minimal structure of a theory must provide objects (things) and relations between them in order to be thinkable by the human intellect.
To ensure consistency, category theory supplemented with locality is used as a mathematical framework. By this means, relations are regarded as directed and may be composed to form new relations. Furthermore objects and their relations may act as objects themselves, called *composite objects.*

A General Interdisciplinary Language of Science

It is noticed that the assumptions are not specifical physical assumptions, which allows other sciences to be described by this theory as well. Chemistry is one of the few disciplines that makes use of a structural representation (in terms of structure formulae), and might therefore easily be embedded into this framework.
It is possible that theories of different disciplines have certain aspects in common, which are not recognized because they are formulated in different languages. Sharing one formulation, these common aspects may be revealed. This leads to further abstraction and again helps to understand nature a little better.

1.1 THE CONCEPT OF COMPOSITE OBJECTS

The idea of composite objects is very old. Houses are built of bricks, sentences are built of words; also galaxies consist of stars, solar systems and black holes, molecules consist of atoms and so on.
As these examples show, this concept may be used in a constructive as well as a descriptive way, which allows to view large systems on different scales. A naive thought suggests that the reason for introducing different scales lies in the incapability of handling the vast amount of information at the most microscopic level. The following examples show, why this is not the case.
The human mind has to deal with objects at different scales in everyday life: The job of an orchestras director is questionable, unless the orchestra is accepted as the directors musical instrument.
Chinese characters, as another example, are composed of a fundamental set of characters, called radicals. In contrast to the musicians, the composite chinese characters often obtain a different meaning when combined. The same happens on a even coarser scale: Combining composite characters, they again obtain a different meaning (see figure 1.1).

Ⓐ 日+木 - 東

Ⓑ 東+西 - 東西

Figure 1.1: *A)* The chinese radicals *ri* and *mu* (meaning *sun* and *tree*) are composed to the character *dong* (*east:* Where the rising sun can be seen through the trees). *B)* This word for *east (dong)* combined with the word for *west (xi)* used as a composite word means *object* (everything from east to west). One cannot understand a chinese sentence only by translating the single characters!

How the Concept of Composite Objects Implies Multiscale Phenomena

Composition of objects is always accompanied by introducing new scales. Describing a real system one may ask how composite objects are identified, or: What makes a composite object an object? The answer of course lies in the objects interaction with its environment, the relations to its neighbours.

A composite object is identified as an object, when it behaves like an object.

While describing real systems, this identification is done by the observer, which introduces cognition in a natural way: *Composition of objects can be considered as a cognitive process.*
Furthermore it is important to notice that relations between composite objects may be of a new kind. As an example from physics, a bunch of molecules may be considered as a gas (identified as such, because it differs somehow from other gases). While molecules differ from each other by properties like position in space and momentum, gases differ in volume, temperature, pressure, etc. The crucial point is that these gas properties, their relations and their dynamics are not defined at the microscopic level (just like the *flying bird* from the prologue). As a result, the same processes can be described at two scales (see figure 1.2). Which one to use depends on the question under consideration.
Another example is well known to everyone who works with computers - a system crash. The objects at the microscopic scale are electrons, wires, transistors, and so on. At the macroscopic scale one has the operating system that reacts to certain input (see figure 1.3). A system is said to crash, if it does not respond to *the user's* input the way *the user* expects it to. Different situations are possible: If the screen turns black, if the

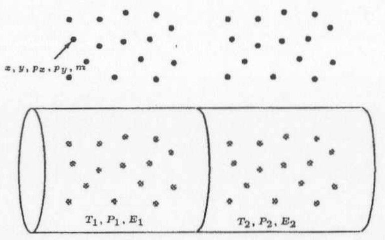


Figure 1.2: A gas viewed at the microscopic and a macroscopic scale: The relevant objects have different properties at different scales.

screen freezes, or even if the mousepointer still moves, but does not react to mouseclicks, the system is said to crash. In such a case, the transistors and microchips still work fine. A defect can definitively not be seen at this scale. Why? Because the term *crashed system* is brought up by the user, who does not even necessarily know anything about the inner life of a computer. It is more or less another term for: *The user cannot use the system in this state* and rather has to do with the relations between the user and the machine than with transistors and electrons.

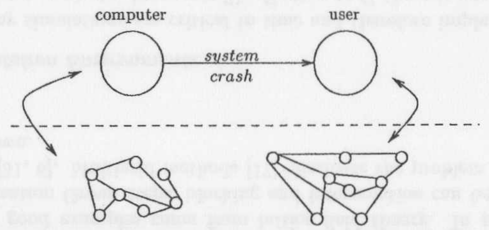


Figure 1.3: Computer and user viewed as composite objects and their relation.

1.2 Modeling Complex Systems

During the recent years more and more problems are being tackled with the help of computers, scientific as well as economic. The increasing development of technologies allows to produce extremely complex problem scenarios which require just as clever and elaborate strategies to solve. The progress of technology seems to be too fast for the work on the solutions to cope with these.
This is shown very impressively by the development of the Denver International Airport [9]. The construction of this 4.2 billion dollar project was scheduled for the years 1989 until the end of 1993. To navigate this extensive system, a very complex computer system was developed. Take off and arival of airplanes were to be managed simultaneously. 193 million dollars were spent just for the the luggage transportsystem. Laserscanners to detect luggage codes automatically, thousands of photocells to monitor the wagons on a 21 km railway system were to be controlled by 300 computers.
The day the airport was going to be put into operation, a series of catastrophies occured: Luggage wagons were not transported correctly, luggage was damaged or got lost. The airport was finally closed again and opening had to be postponed by more than one year. The consequence was a loss of 1.1 million dollars per day.
The main reason for this catastrophy was found in the software system responsible for the luggage transportation. The complexity of todays software grows so rapidly that its design and analysis becomes a very important issue.

The occurrence of this desaster demonstrates the importance of elaborate modeling techniques. The fact that it took such a long time to find the bug on the other hand reveals the lack of appropriate analyzing techniques. Both is being worked on hard during the recent years.

1.3 Numerical Simulations in Physics

"For a theorist studying dynamical critical behaviour, [Critical Slowing Down] is a fascinationg phenomenon; but for a practicioner of Monte Carlo, it is a pain."

A. D. Sokal[23]

Many physical problems involve a very large number of degrees of freedom. On the other hand, many theoretical methods apply best to models with

independent variables, being able to break down the problem to problems with only one degree of freedom [29] (like separating variables to calculate a Schrödinger equation).
One way to handle systems with many degrees of freedom is to run computer simulations. Integration can be performed numerically without the need for analytical approximations. Interesting but intractable terms don't have to be omitted from models at the beginning of the modeling phase, further more visualization techniques illustrate the structure of complex models [8].
Because of the discrete character of the computer memory, discrete processes are candidates for numerical simulations rather than continous ones.
Obviously good examples come from lattice field theory. In performing Renormalization Group steps, blocking and interpolation can be done numerically [31, 6]. Multigrid methods [17] eliminate the problem of critical slowing down.

Implementation Environments

While many simulations are critical in time and therefore implemented in hardware-near computer languages like *Fortran* or *C*, there is a tendency to use object oriented techniques, which correspond to the theory of complex systems in a natural way. Since processors become still faster, and the demands on simulations grow, using object oriented methods seems to be a good way to handle the increasing complexity. K. Wilson already mentioned the possibility to improve algorithm design using object oriented techniques [30, 24]. Unfortunately the lattice field theory community did not take much notice of this advertisement until 1993, when code for Monte-Carlo simulations of $SU(N)$ lattice gauge theories was translated to the object oriented programing language *C++* by Moriarty et al. [20, 24].
For the simulation tool, which is introduced in this work, the language *C++* is used. This allows to implement the systems theory's modular concepts in a quite natural way.

Chapter 2

The Theory of Complex Systems

The theory of complex systems provides the basis of this work. The main concepts and the mathematical formulation are pictured in this chapter.

2.1 Introduction

As argued in the introduction, a general theory should have the least amount of a priory structure possible. The system theoretical approach is to describe complex systems as networks of objects related by directed arrows.
What kinds of physical systems fit into this concept? Systems discrete in time and space are suitable. Each coordinate-point in space may be described by an object. A lattice field might again be described by a lattice of objects containing the field values, where the relations between these objects might carry matrices describing parallel transporters (see figure 2.1).

In order to be exact, a mathematical framework is needed.

2.2 Mathematical Foundation

To describe a system formally, a set of objects $X, Y, ...$ and a set of directed arrows, representing the relations, $f, g, h, ...$ are needed. An arrow f pointing

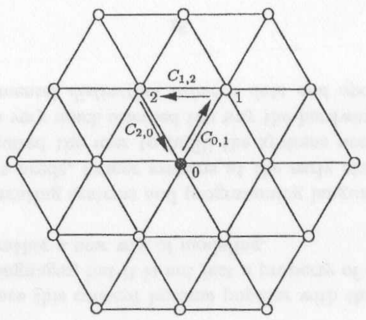


Figure 2.1: Curvature in spacetime: The paralleltransporter is obtained from composition of links.

from an object X to an object Y will be denoted by: $f : X \to Y$.
Furthermore the following properties will have to be fulfilled [16]:

1. **Composition:** Given two arrows $f : X \to Y$ and $g : Y \to Z$ there exists an arrow
$$g \circ f : X \to Z \tag{2.1}$$
This composition is associative. $h \circ (g \circ f) = (h \circ g) \circ f$

2. **Adjunction:** For each arrow $f : X \to Y$ there exists a unique adjoint arrow $f^* : Y \to X$. The adjoint of the adjoint of an arrow is the arrow itself $f^{**} = f$ and the adjoint of a composed arrow is
$$(f \circ g)^* = g^* \circ f^* \tag{2.2}$$

3. **Identity:** For each object X there exists a unique arrow $\iota_X : X \to X$ with the property
$$\iota_X{}^* = \iota_X \tag{2.3}$$
and for $f : X \to Y$
$$\iota_Y \circ f = f = f \circ \iota_X \tag{2.4}$$

4. **Locality:** Some of the arrows other than identities are declared *fundamental* and also called *links*. All other nonidentity arrows f can be made from *links* by composition and adjunction
$$f = b_n \circ ... \circ b_1 \tag{2.5}$$

where b_i are links or adjoints of links.

A *distance* bewteen two objects is defined via the minimum number of links needed to compose an arrow between these objects. The *neighbourhood* of an object contains the object itself, the objects at a distance of one link and the links connecting them.

5. **Composite Objects:** A triple (Ω, A, Λ) of a set of objects Ω, a set of arrows A, where $A \ni a : \Omega \to \Omega$ and a subset of fundamental arrows $\Lambda \subset A$ is called a *system*. An object $o \in \Omega$ can contain an inner structure and therefore itself be a system, called a subsystem or a *composite object*[1].

Dynamics

The dynamics acts locally on systems: The rules to manipulate a system apply to an object within a neighbourhood, i.e. no information from outside the neighbourhood is used. Manipulation is done mostly by reassigning the fundamental arrows. This way, motion on a one dimensional lattice space can be treated as a fundamental rule in terms of composition of arrows, see figure 2.2.

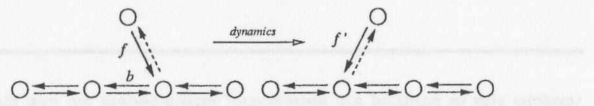


Figure 2.2: Motion: The solid links with their adjoints represent neighbourhoods between space points. The dashed arrow is not fundamental. Its adjoint is a link establishing the relation of the object (above) and its position in space. A local rule assigns a different set of fundamental arrows, allowing the interpretation of a change in the objects' position in space ($f' = b \circ f$).

Besides reassignment of fundamental arrows, birth and death are two important manipulations: Objects (and arrows) can be copied, and destroyed. This is a very essentail point, in which systems theory differs from many other network theories with a constant number of objects.
Further studies can be found in references [27].

[1]An iteresting analytical discussion on the concept of composite objects can be found in reference [27]

Chapter 3

The Object Orientation Paradigm

This chapter shall introduce the concepts and terminology of the object oriented approach as well as its relation to the concept of composite objects.

3.1 The Object Oriented Approach

The idea of modularization has influenced many sciences during the recent decades. Dividing problems in smaller subproblems, has proved to be an efficient strategy. Multigrid methods in numerical physics or separation of variables in calculus are good examples.

In computer science this concept became popular with the introduction of object oriented languages, but it is not just a property of certain programming languages, rather a new way of modeling.

While current operating systems and programming languages are designed to fulfill the users needs, former systems at the early stages of computer programming required the user to fulfill the systems needs. The way of programming was very much oriented the way the hardware worked, which required a fundamental *distinction between data and operations on data.*

Data is static unless it is changed by an operation, while operations have no state of their own and are only used to affect data.[21]
During the recent decades, computer hardware became faster, less expensive and more capable of handling large programs, which required more modular programming concepts.
Modern technologies make it possible to take the programmers needs into account. And since the separation of data and operations is somehow unnatural, the object oriented approach unites the operations with the data they operate on.

Objects and Classes

In object oriented computer languages, such a unit containing data and operations is called an **object**. These objects are treated as black boxes. One does not (and *should* not!) have to know the details an object carries, rather than how to communicate with it. For that purpose, the object provides an interface, which consists of operation declarations, other objects can call to invoke an operation. This interface can be viewed as the surface of a black box, while the objects' implementation details are inside (see figure 3.1). A criterion for good object oriented programming is to decide what to put into the interface and what belongs inside.

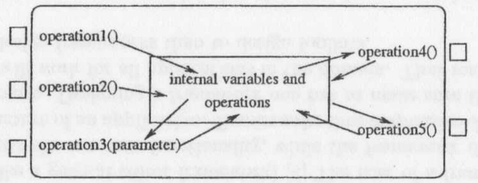


Figure 3.1: The contents of an object. The small boxes represent the interface.

A **class** can be seen as a template that holds the abstract definition for one or more objects, which are said to be **instances** of that class. Such a definition consists of *declarations* of data and operations on one hand, and the *implementation* of the operations on the other hand. It makes sense to distinguish between declaration and implementation, since only the declaration provides the interface for the classes instances: One cannot see, how a certain operation is implemented, only the name of that operation, and what kind of parameters it needs!

The data and operations are referred to as **data members** and **memberfunctions** or **methods** respectively. The declaration of a memberfunction, i.e. its name, its parameters and the type of its return value, is called **signature** of that function. Finally, an **interface** is defined as the sum of all signatures of a class.
Within a class definition, data members and methods can be declared **private** and **public**. Data members and methods declared public build up the interface and are thus accessable by other objects, while declared private they can only be used by method implementations within the same class[1].

Encapsulation

The advantages of modularity have been mentioned before. Large problems are easier to handle, once they are broken down into small parts. Thus, one criteria for good object oriented software design is to separate different parts of the model as much as possible. Among other things it is helpful to make data members accessable *only* through methods. The internal state of an object is then said to be **encapsulated**. This way, one has control, where data is changed: Only within the implementation of the corresponding class[2].
As an example, a program might solve a problem numerically, by applying different arithmetic operations to objects of the type *Real Number*. It might then be desirable to extend the application to complex numbers. With structured programming methods, the multiplication rule for complex numbers had to be inserted any place, a multiplication occurs. With object oriented methods, only the declarations had to be changed. The objects can be instantiated from any class as long as the class provides a multiplication within its interface. This separates the abstract multiplication occuring in the program from the concrete implementation.
Note that the concept of encapsulation is consistent with the systemtheoretical concept of composite objects!
Once a small but common problem is solved, the software can be used by

[1]They can also be declared **protected**, meaning that child classes (s.b. *inheritance*) have access to them, while declared private, child classes do not.
[2]It makes it a lot easier to answer questions of the type: *I wonder where I set this variable to zero!*

programs of all different kinds. This is of course very efficient. The code can be reused in two ways: As *Toolkits* and as *Frameworks.*

Code Reuse and Design Reuse

A **toolkit** is a set of related classes that provide general-purpose functionality (like hardware input/output libraries or a set of list classes) [4]. It emphasizes *code reuse.* A toolkit has to be designed carefully, because it has to work in many different applications to be useful.
A **framework** might be viewed as the complement of a toolkit: It consists of cooperating classes that provide a reusable design for a specific class of software (like a general editor framework) [4] The user of a framework has to implement the concrete functionality, while the framework dictates the overall structure of an application. Frameworks thus emphasize *design reuse* over code reuse. Designing a framework one has to make sure that the architecture will work for all applications in the domain. That makes it even harder to design frameworks than to design toolkits.

The simulation program, introduced in the next chapter, can be considered as a general simulation framework. By adding new code (i.e. composing *enzymes*[3], or even writing new enzymes in C++), this framework is adjusted to the concrete problem that is to be simulated. Also in this case, the subtle point is to make sure it can be used for all kinds of simulations. The answer of course is found in the absence of a priori structure in systems theory.

3.2 INHERITANCE

To push the concept of separation even further, **inheritance** - another concept of the object oriented approach - is introduced, in order to separate an interface from its implementation.

A class can be defined as a *specialization* of another class that already has been defined [27]. This new class **inherits** all the properties and functionality of its **parent class** (i.e. it contains all data members and methods the parent class contains). Additionally it can be extended by new properties

[3]An enzyme is an object responsible for a fundamental dynamical action on the system. See next chapter.

and functionality. The parent class is also called **baseclass**, and the child class is said to be **derived** from its baseclass.
Modeling a chess game one might write a class *chessman.* A property of a chessman will certainly be the 2-digit position on the chessboard. The individual types of chessmen (*king, queen, bishop,* etc.) might be defined in classes derived from the class *chessman* (see figure 3.2). These classes might provide operations that determine which moves are legitimate.

Polymorphism

Polymorphism is a concept which ensures that a call of a method reaches the corresponding implementation within the class hierarchy. A child class derived from a class having implemented a method may as well implement a method with the same signature. This is called **overloading**. An object can be declared as of the type of the baseclass, but instantiated from the child class. This is very useful to handle abstraction, programming to an interface rather than to an implementation. In this case, calling the overloaded method, it is not trivial for the compiler to decide whether to perform the baseclass' implementation or the overloaded implementation of the child class.
In *C++*, methods can be declared either *virtual* or not: A **non virtual** declaration of the method always results in performing the baseclass' implementation, where a **virtual** declaration results in performing the child class' implementation.
Some object oriented computer languages enforce virtuality, stating that non virtual declarations are not truely object oriented and lead to programming bugs.

Interface and Implementation

So far, there are two views on inheritance: *Specialization* and *code reuse.* Code reuse can always be achieved by inheritance. But Inheritance does not necessarily mean specialization. Consider a class *square* [22]. It has one real value as its property, describing the length of a side. In order to reuse code, one might define a class *rectangle* by deriving from the class *square* and *extending* it by another real value to describe the other side of the rectangle. Of course a rectangle is not a specialization of a square (rather the other way around). Therefore one should not use inheritance in this case.

Favour specialization over code reuse

In clean object oriented design, one distinguishes between *abstraction* and *extension*, abstraction implying specialization and extension implying code reuse. These two ways to view inheritance have to be treated differently in softwaredesign.
While the concept of extension (or code reuse) makes perfect sense when large amounts of code can be kept centralized in one class, it has to be treated extremely carefully. From its structure it is not as natural as the specialization concept, and thus often leads to programming bugs.

Abstract and Concrete Classes

An **abstract class** is a class no instances are created from. In the chess

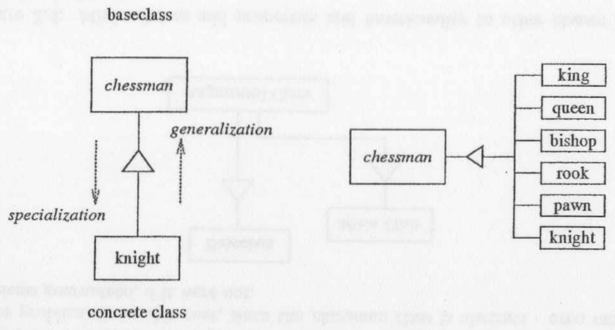


Figure 3.2: Inheritance: The arrow points always to the baseclass, which is set italics, if it is an abstract class. The knight is a special case of a chessman, and the chessman is an abstraction of the knights, rooks, bishops, etc.

example, instances of the class *chessman* do not make sense. The *chessman* class is a good example for an abstract class, it only serves as a class to derive from (see figure 3.2). The derived classes, implementing the methods declared in the abstract class, are then called **concrete classes**.

An abstract class only provides an interface for other classes.

Within the program, the interface of the abstract class can be used, without knowing the concrete class an object is instantiated from.

In the chess example, one could request the position of a chessman-object, not knowing if it is a *king*, a *knight* or any other chessman. Next, one could ask whether a move (e.g. invoked by the user) is legitimate for an object, neither knowing the concrete object nor the rules for that object. This task is done by the object itself: E.g. a *checkMove()* method is implemented differently for the *knight* than for the *rook* and so on.

Multiple Inheritance

It is thinkable that a class inherits its properties or interface from more than one class. There are different opinions on multiple inheritance and it has always been a hot topic whether it should be allowed for clean object oriented programming at all. Some computer languages, like *Java*, do not support multiple inheritance, while *C++* for example does.
The problems of multiple inheritance are related to the problems of code reuse: Each class has a special method called **constructor** which is called when an object is instantiated. Deriving from two classes, which constructor should be called first? This easily leads to programming bugs, which can be avoided by implementing an empty constructor.
Now, if these two classes are derived from the same baseclass (see figure 3.3), operations implemented in its constructor are performed twice, where only one object is created.
In the chess example, one could argue that the possible *Queen's* moves are the sum of the possible *Rook's* moves and the *Bishop's* moves. Thus it makes sense to derive the *Queen* class from the *Rook* class and the *Bishop* class. Now, the constructor of the chessman class creates and initializes a field to store the position of that chessman. Not taking care of that, the new defined *Queen* ends up with two positions on the chessboard (see figure 3.3). This is of course nonsense.
To avoid these problems, abstraction is the right way out. If the constructors are empty, nothing will be done twice. Thus as a statement to the multiple inheritance discussion, one does not necessarily have to renounce multiple inheritance, as long as one keeps in mind:

At most one of the baseclasses may be a concrete class

The abstract classes can be understood as classes adding interfaces for further functionality to a class.
One could provide functionality to several classes by having abstract **Mixin classes**, implementing certain methods and properties [4], (see figure 3.4).

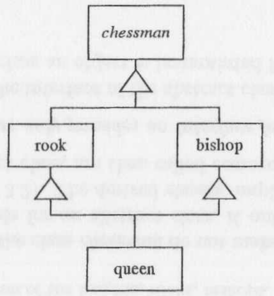


Figure 3.3: Multiple inheritance: This structure is dangerous: Operations common to different chessman will be performed twice for the queen. Note that the cycle is not problematic in this case, since the *chessman* class is abstract - even more problems guaranteed, if it were not.

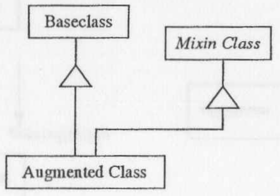


Figure 3.4: Mixin classes add properties and functionality to other classes by multiple inheritance.

Making use of this, one can try to fix the chess example in the following manner: Mixin classes *f-Knight*, *f-King*, *f-Bishop*, *f-Rook* and *f-Pawn* are derived from an abstract class *chessmanlyness.* They only implement the method which verifies moves on the chessboard. A class *chessman* then implements the position and anything memory is needed for. The concrete chessmen are implemented by multiple inheritance: The *knight* is derived from the classs *chessman* and *f-Knight*, and so on. Only the *queen* is derived from three classes *chessman*, *f-Bishop* and *f-Rook* (see figure 3.5)[4].

[4]Of course there is another catch to it: The rook can do a rochade, which the queen is not supposed to do.

Having code reuse in favour, this would be the right solution. On the other hand, the structure has become rather complex, to keep a clean design. In this case, code reuse should be neglected in favour of a less complex structure, and a even better reason against such an implementation is the fact that a queen simply *is not* a rook, combined with a bishop.

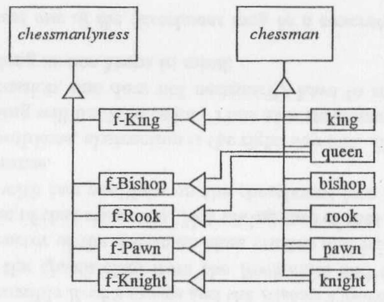


Figure 3.5: Multiple Inheritance

3.3 From Flowcharts to Class Diagramms

It always seemed desirable to have a diagramatic representation of complex programs in order to reveal coarse mistakes. The importance of a good visualization was realized from the very begining of computer programming. In the early days this was achieved with the help of flowcharts: Graphical symbols have been assigned to different program steps, while evolution in time has been visualized by arrows connecting these symbols (see figure 3.6). When programs became complex, ***structured programming*** became necessary. Drawing flowcharts was one way to increase structure, another very important point was to avoid[5] leaving the program flow without return. The program should always come to a finish point. Another kind of diagram was introduced (Nassi-Schneidermann[10]) to take care of that problem (3.6).

[5]Gain of structure is usually achieved by prohibition rather that recommendation of programming style[4].

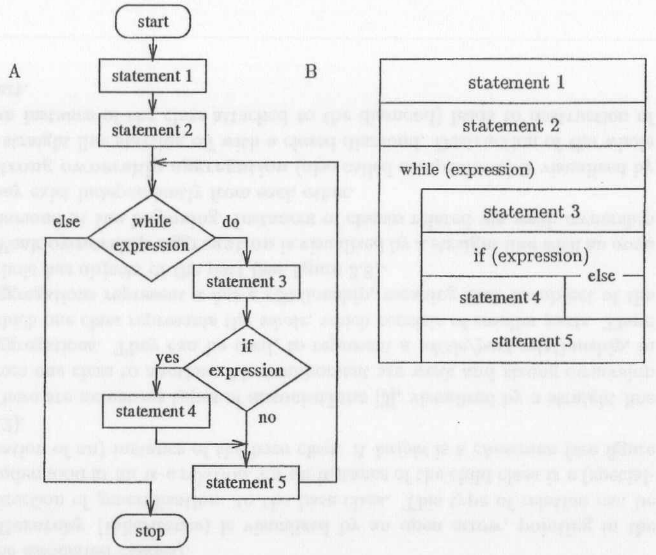


Figure 3.6: *A)* A typical flowchart diagram for a program containing one loop and an if clause. *B)* A Nassi-Schneidermann diagram for the same program.

Though these diagram types emphasize the benefits of structured (sequencial) programming techniques, they are also limited to these. With object oriented languages, a more modular visualization is needed.
In order to understand both the structure and the function of the objects involved in a complex software system, it is impossible to capture all the subtle details in just one view [2]. Different types of diagrams are used to resemble structures of class objects, inheritance mechanisms, individual behaviour of objects, dynamic behaviour of the system as a whole, etc. Each diagram type can be understood as a projection of the whole system.
Unfortunately no standard exists, describing the symbols to use for object oriented models, though different descriptions only differ in detail. The notation chosen here is the one used for the *Unified Modeling Language* (UML) [3].

The UML suggests a vast variety of diagram types to cover all properties of the system from different points of view. They can be divided into diagrams describing the static parts of a system:

- Class Diagrams
- Object Diagrams
- Component Diagrams
- Deployment Diagrams

and those describing the dynamic parts of a system:

- Use Case Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- Statechart Diagrams
- Activity Diagrams

In this work, only three of these are presented. Good descriptions of the remaining diagram types can be found in the references [3] and [27].

3.3.1 Class Diagrams

Most important for this work are class diagrams. They show the relationships between classes and between instances of them.

Classes

A class is visualized by a rectangle, including the class name, as well as data members and methods relevant for the context of the diagram (see figure 3.7 and 3.8).

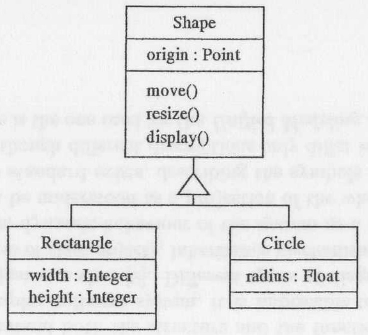


Figure 3.7: A base class *shape* and two classes with data members and methods.

Associations

There are two kinds of relationships: Hierarchy (representing the hierarchy of classes) and association (representing relationships between instances of the associated classes).
Hierarchy (inheritance) is visualized by an open arrow, pointing in the direction of generalization: to the base class. This type of relation can be understood as an *is-a* relation, for an instance of the child class *is a* (specialization of an) instance of the base class: A *knight* is a *chessman* (see figure 3.2).
There are numerous types of **associations** [3], visualized by a straight line from one class to another. Most important are weak and strong ownership aggregations. They can be used, to represent a *whole/part* relationship, in which one class represents the *whole*, which consists of smaller *parts*. These aggregations represent a *has-a* relationship, meaning that an object of the whole *has* objects of the part (see figure 3.8).
Weak ownership aggregation is visualized by a straight line with an open diamond at the beginning. Instances of classes related via weak ownership may exist independently from each other.
Strong ownership aggregation (also called composition) is visualized by a straight line starting off with a closed diamond. Destruction of the whole (an instance of the class attached to the diamond) leads to destruction of part.

It may be useful to diagram when instances of a certain class are able to create instances of another class. This is called **realization** and visualized by dashed line with an open triangle at its end.

Further it might be interesting for any kind of association, to know how many instances of one class is associated with how many instances of the class it is associated to. This is done by specifying a **multiplicity** number to each end of an association, where "*" or "0..*" means zero or more, and "1..*" means one or more (see figure 3.8).

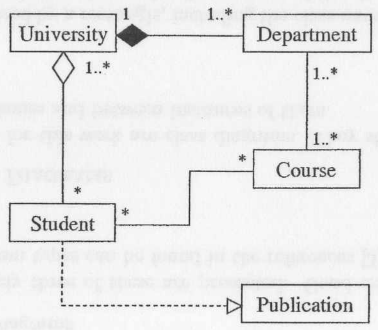


Figure 3.8: A very brief model to illustrate the different kinds of associations: The university has one or more departments, as well as one or more students. The departments cease to exist when the university gets closed, while the students of course live on.

3.3.2 Object Diagrams

Though class diagrams show the relationships not only between classes but also between their instances, they do specify how many instances are actually created at what point of time.
Object diagrams show a set of objects and their relationships at one point in time [3]. Like class diagrams they address the static design view or static process view of a system, but from the perspective of real cases. In this sense, object diagrams can be understood as instances, or *concrete applications* of their abstract class diagrams.

Object diagrams commonly contain **objects** and **links**.
Objects are visualized by rectangles, containing the object name, separated by a colon from the class name, it is instantiated from, and both names underlined. If necessary, the objects data members and their values can be added below, separated by a horizontal line (see figure 3.9).
Links are visualized by straight lines, connecting two objects (see figure 3.9).

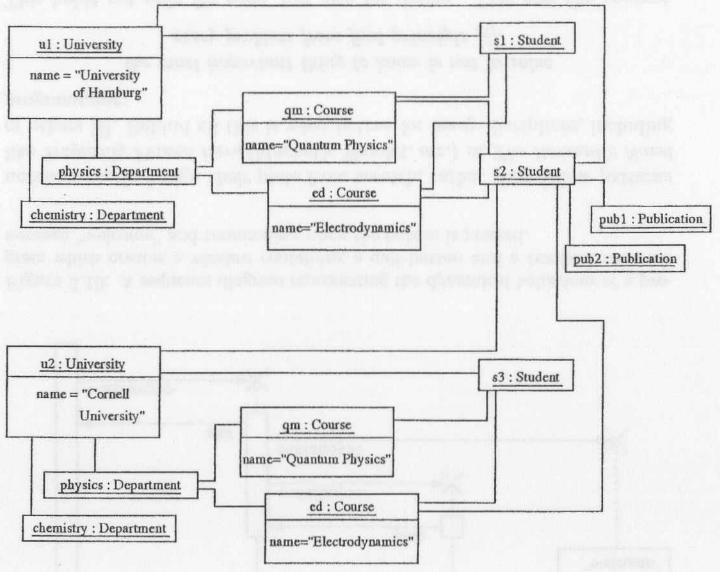


Figure 3.9: A typical object diagram for the above class diagram.

3.3.3 Sequence Diagrams

As a diagram type, visualizing a systems behaviour, sequence diagrams show the objects lifetimes, and their dynamical interactions.
In object oriented thinking, invoking an objects operation is often viewed as *sending a message to an object*. This emphasizes the objects picture of

a black box even more. So sequence diagrams describe a set of objects and the time ordering of messages sent and received by them.
This is visualized in a table that shows objects arranged horizontally and messages, ordered in increasing time vertically [3]. An objects lifetime is presented by a vertical dashed line. Messages are presented by arrows starting at the object sending out the message and ending at the receiving object. Typically these arrows are drawn horizontally, since sending messages usually does not take much time. In special cases though, they may be drawn at different angles to visualize duration.
During focus of control, i.e. when an object sends or receives a message, a rectangle is shown, instead of the dashed line (see figure 3.10). Objects may exist from the very beginning to the end of the entire process, these are drawn at the top of the diagram, with dashed lines drawn to the very bottom. Objects may also be created during the process. Their lifelines start with the receipt of a message named <<*create*>>. They might as well be destroyed, receiving a message <<*destroy*>>. An x is placed at the end of the lifeline.
When the state of an object changes, its icon is drawn again at the right vertical position, containing the new state (see the *Textfield* object in figure 3.10).

3.4 Design Patterns: Object Orientation at a Higher Level

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same time twice."

Christopher Alexander[4]

The fact that Christopher Alexander used this quotation describing patterns in buildings and towns, although it fits to the context of software design as well, shows that design patterns are part of a more abstract concept. In many constructive processes patterns evolve quite naturally: What distinguishes the experienced chess player from the novice is the repertoire of chess techniques, which one to use depending on the chess partner. Experienced

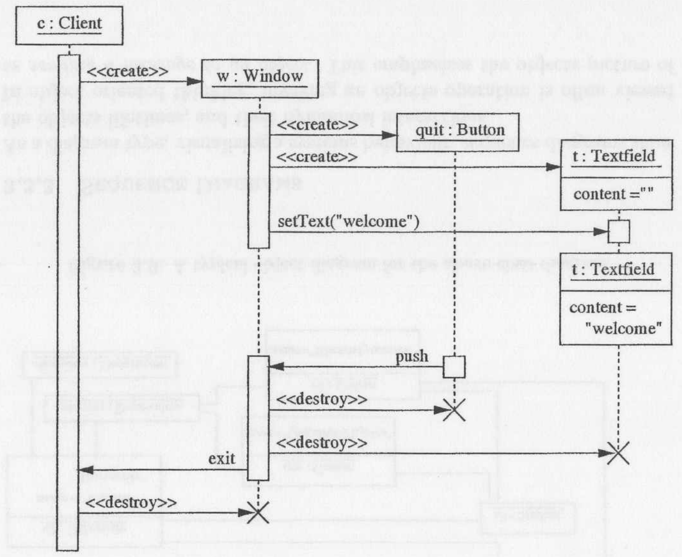


Figure 3.10: A sequence diagram representing the dynamical behaviour of a program which creates a window containing a quit-button and a textfield with the message "welcome" and terminating when the button is pressed.

novelists rarely design their plots from scratch, rather they follow patterns like *Tragically Flawed Hero* (Macbeth, Hamlet, etc.) or *The Romantic Novel* or others [4]. Behind all this is what is true for many disciplines, including programming:

the most important thing to know is not to solve every problem from first principle [4].

This holds not only for code, but also for design. This way the concept of code reuse is extended to design reuse, which is independend of concrete code, even of the computer language: Design patterns are often expressed in terms of diagrams. The most popular ones are given names, so programmers can refere to them uniquely. Furthermore it must be noted to what kind of problem they can be applied to and what consequences they bear.

Each programmer usually has many patterns in mind, based on his programming experience.
Once having chosen a design pattern, many decisions are already taken. To decide which pattern to use is therefore very subtle. As a consequence in the case of a bad choice problems might occur, often resulting in inflexibility.
If for example the operations of a software are implemented in member functions of all kinds of classes, implementing an *Undo* function is a very tough and complex job in this situation. In a redesign, operations could be encapsulated inside classes derived from an abstract class *GeneralOperation*. This way, it can easily be kept track of all operations that are performed, and it is quite easy to implement even a multilevel-level *Undo* function. This idea may be kept in a pattern which addresses problems where operations shall be reversible.
Two very commonly used patterns are introduced below. More of them can be found in [4].

The Bridge Pattern

In many cases multiple use of inheritance leads to multiple implementation of the same code. This happens, e.g. when inheritance is used to specialize with respect to one property, and used again on the child classes to specialize with respect to an independend second property.
Resuming the above example of a chess program, it might be useful to have a graphical as well as a textual presentation for each chessman. The inheritance tree branches once to specify the kind of chessman and again to specify the presentation. It does not matter which specification is done first (see figure 3.11): Each chessman has to be implemented twice, once in each presentation.

This problem often occurs in context with technical aspects (e.g. when more than one graphicslibrary is used). The solution to this is given by the *Bridge Pattern*, which addresses the problem by putting the two specifications (like the chessman abstraction and the presentation) in separate class hierarchies. This gives the impression of something like an *abstract implementation* (see figure 3.12).

The Abstract Factory Pattern

The great benefit of virtual methods is the ability to treat objects of different classes the same by using the interface of their baseclass. The decision

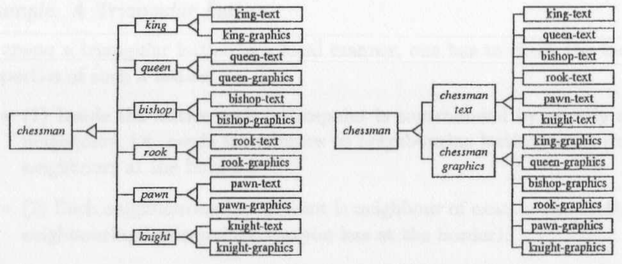


Figure 3.11: Inheritance is not always the most flexible solution: Each chessman has to be implemented in a textual and a graphical version.

which concrete class to use still has to be made somewhere: At the point where the object is instantiated. Due to the lack of a virtual constructor, there is no way to avoid it.

Now it is often desirable to make a decision only once, e.g. at the startup of the program (which might be a decision concerning the kind of presentation, or what language to use to communicate to the user).

The *Abstract Factory Pattern* shows the way out of the dilemma: One object, the **factory**, is used to instantiate all objects concerned by this decision. Their instantiation occurs at no other place outside the factory. Where such an object shall be created, a message is sent to the factory to invoke the desired instantiation. This way, the instantiation of an object is kept inside a method of a factory. These of course can be virtual, and are often referred to as *virtual constructors*.

Thus, an **abstract factory** provides the interface containing one factory method for the instantiation of each class concerned. **Concrete factories** are then derived from it, to implement the instantiation of the different concrete classes.

This pattern is used by the SyCL presentation, see figure 4.7. The decision, whether a graphical or a textual presentaion shall be used is made at the startup of the program via a command line parameter.

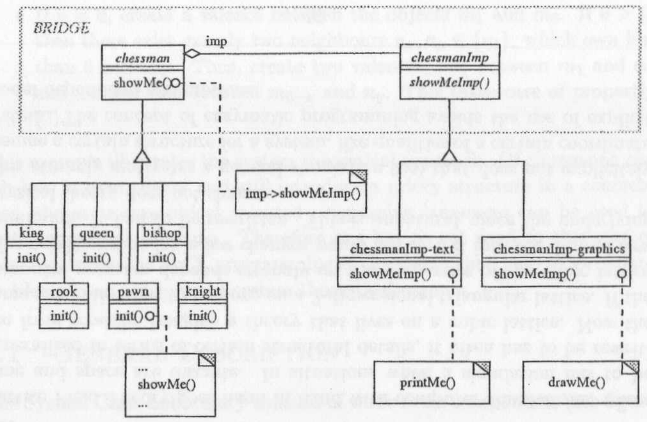


Figure 3.12: The *Bridge Pattern* keeps the chessman abstract and the presentation in separate class hierarchies. The boxes connected by dashed lines show some important implementation details: An object of the abstract class *chessman* has access to an object of the abstract class *chessmanImp*. Implementing the *chessmans* method showMe() via imp→showMeImp() means that each call of the *chessmans* showMe() method is delegated to a *chessmanImp* object in order to perform its showMeImp() method. And since this is virtual, the showMeImp() method of the concrete *chessmanImp* implementation is called.

3.5 Finding Appropriate Objects

The hard part about object oriented design is to decompose a system into objects. There are many different approaches. Modeling a part of the real world, the objects found during analysis can be translated directly into design. Dealing with an abstract system, one could write down the problem in sentences, single out the nouns and verbs and write classes to capture the nouns and methods to capture the verbs. One could also focus on the collaborations and responsibilities in the system [4].

Other, more technical criteria like encapsulation, granularity, dependency, flexibility, performance, evolution, reusability are often conflicting, where design patterns again help to make coarse decisions.

Though many objects come from the analysis model, many reliable design patterns involve classes that have no counterparts in the real world. *Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's [4].* Introducing abstractions during design are key to making design flexible.

Chapter 4

The System Class Laboratory

The System Class Laboratory (SyCL) serves as a tool to simulate dynamical behaviour of complex systems. After a short introduction this chapter gives a description of the software. The implementation of the dynamics are discussed in the sections about enzymes and predicates. Object oriented methods are pointed out in the sections about the core library, the graphical presentation and the language interpreter. This chapter concludes with a few words on coding systems.

Lattice Field Theory goes hand in hand with computer simulations, where time and space are discrete. In situations when a simulation has to be generalized in terms of certain structural details, it often has to be rewritten from scratch: Imagine a theory that lives on a cubic lattice. Now the same simulation has to be done on a 2-dimensional triangular lattice. If the computer program depends strongly on the properties of the cubic lattice (like neighbourhoods), a few changes would not turn it into the desired new simulation. It had to be rewritten. This is unnatural, since the underlying physical theory does not change.
This strongly motivates a general simulation tool that does not explicitely assume a certain structure for a system, like qualities of a certain coordinate system. The concept of enzymatic programming avoids the use of explicit model dependend assumptions.

Example: A Triangular Lattice

To create a triangular lattice in a local manner, one has to check the local properties of such a lattice:

- (1) Inside the lattice each latticepoint is sourrounded by exactly six neighbours, i.e. needs six valences to neighbouring latticepoints, (less neighbours at the border.)
- (2) Each neighbouring latticepoint is neighbour of exactly two further neighbouring latticepoints, (maybe less at the border).

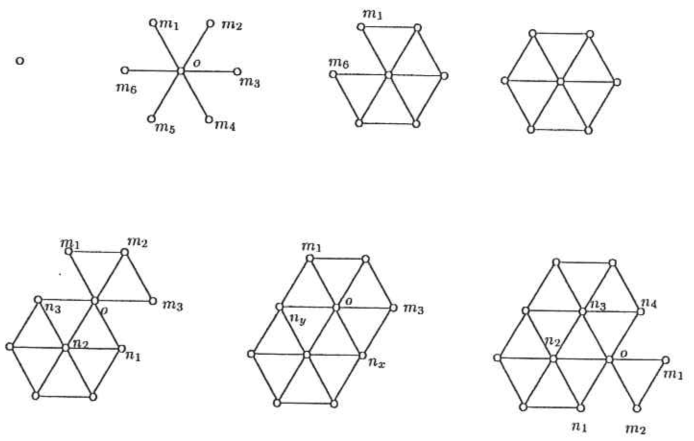


Figure 4.1: Creation of a triangular lattice.

A system theoretical description of such a construction might look like this (see figure 4.1):

- For a dynamics of n steps, apply n times the following rule to each object of the system, starting with one root object.
- Let o be an object and v be the number of its valences to its neighbours $n_1, ..., n_v$. If $v < 6$, create $6 - v$ objects $m_1, ..., m_{6-v}$, $6 - v$ valences between the objects m_i and o and $6 - v - 1$ valences among the objects m_i in the order of their creation. This takes care of property (1).

- If $v = 0$, create a valence between the objects m_1 and m_6. If $v > 0$, then there exist exactly two neighbours $n_x, n_y \in \{n_i\}$, which own less than 6 valences. Thus, create two valences, one between m_1 and n_x and another one between m_{6-v} and n_y. This takes care of propertiy (2).

This example illustrates the system theoretical approach. Of course one can (ab-)use system theory and still introduce a priory structure in a concrete case, the same way object oriented programing languages can be used to write sequential programs. Therefore one has to get used to this way of thinking, where the SyCL programming environment shall help to formulate problems in an appropriate system theoretical manner.

4.1 General Description

The System Class Laboratory consists of three main components (see figure 4.2):

- *The Core Library* provides a set of classes to run systemtheoretical simulations.
- *The Presentation* establishes the user interface.
- *The Language Interpreter* provides a programming environment which parses commands, executes them and allows an easy setup of systemtheoretical scenarios. It thus serves as an interface between the presentation and the core classes.

The *Interpreter* is a powerful tool, it might be used as an interface to other programs which need to include systemtheoretical components. The *Presentation* is also capable of translating the actions invoked via the user interface into a script which can be read by the interpreter. Operating with a graphical presentation, the user can write programs by using the mouse.

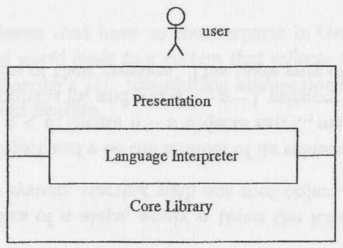


Figure 4.2: The three packages: The core library at the bottom is totally hidden from the user, who communicates with it via the *presentation* package. The *language interpreter* acts as an interface between the core library and the presentation.

4.2 The Core Package

Objects and Valences

According to the theory, the two central classes are those representing objects and arrows. For reasons of effectivity, the environmental structure of an object is separated from its (possible) inner structure. Therefore two classes are needed.
The class **Radical** captures the objects environmental structure, it strongly owns a list of arrows, while the class **Object** carries information about the inner structure, like a complete subsystem or a numerical value[1]. A radical strongly owns an object (see figure 4.3). In system theoretical diagrams, the object-radical separation is ignored. A circle is used to represent an object, possible values are drawn inside.

To run simulations, it is not very efficient to create an instance for every corresponding arrow. Instances of the class **Valence** usually represent the fundamental arrows (see figure 4.3). In special cases the status of a valence can be set to *virtual*[2] representing non fundamental arrows. This is necessary when valences lack fundamental adjoints. For implementational reasons

[1]Numerical values can be understood as abbreviations of subsystems represented by the value in an arbitrary way. The fact that values like transcendent numbers cannot be represented by a finite discrete system does not really restrict the application, since values of transcendent numbers are always truncated in computer simulations.

[2]This *virtuality* of valences in contrast to fundamentality is not to be confused with the virtuality of member functions in context with the object oriented approach.

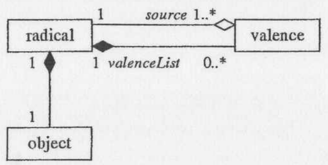
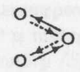


Figure 4.3: A class diagram of the basic components.

valences need adjoints. If they are not fundamental, they are declared virtual. In system theoretical diagrams, fundamental and virtual valences are represented by solid and dashed arrows respectively:



Note that the radical owns the valences, pointing towards it, and the valences own their sources (see figure 4.4). Thus a sweep through the system is performed backwards.

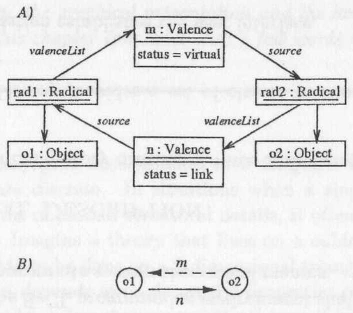


Figure 4.4: *A)* An object diagram of the basic components. *B)* The system theoretical diagram for the same scenario.

As a special case, a valences target may be the same as its source. These kind of valences are called *loops*. Their adjoints are of course loops as well. In special cases, loops can be *selfadjoint*.

Values

Values are implemented via an abstract class **Value**. Among different types of values, most important are those capable of algebraic operations. Derived from the class **Value** the abstract class **AlgebraicValue** declares these operations within its interface and thus ensures correct composition of valences and parallel transportation (of objects along valences). A radical also owns an algebraic value. In contrast to objects, the radicals values are used to store invariants which are parallel transported trivially.

Enzymes and Predicates

Instances of the class **Enzyme** are objects with the ability to manipulate the system locally: They have a method *act()*, which is called with two parameters, an object and a valence[3]. The object specifies the position within the system, while the valence can be viewed as a direction in which to act. Many enzymes do not use this second parameter.
There is only a finite number of fundamental enzymes, also called **microenzymes**, responsible for all dynamic behaviour of a system. All other actions are achieved by sequential application of fundamental enzymes.
As special values, instances of the class **Predicate** can be used to restrict an enzymes action to fulfill a condition. Predicates can be combined logically. A pair (*enzyme, predicate*) is called a **mechanism**.

The system theoretical way of programming is to compose new enzymes by combination of mechanisms.

Mechanism (just as enzymes) take an object and a valence as parameters. Having their action compleded, they also return an object and a valence, e.g. as input parameters for a following enzyme inside a composite enzyme. Most enzymes just pass their parameters to their successors, only few (like the _PRS enzyme) return a different pair. To assure locality it is important that the parameter valence is directed toward the parameter object.

Keys and Membranes

To increase a systems structure, subsystems can be introduced. To separate the inside of a subsystem from the outside, valences can carry *membranes*. The class **Membrane** contains a textual representation of a *key*. Only with

[3]Both are elements from the system may be manipulated by the enzyme.

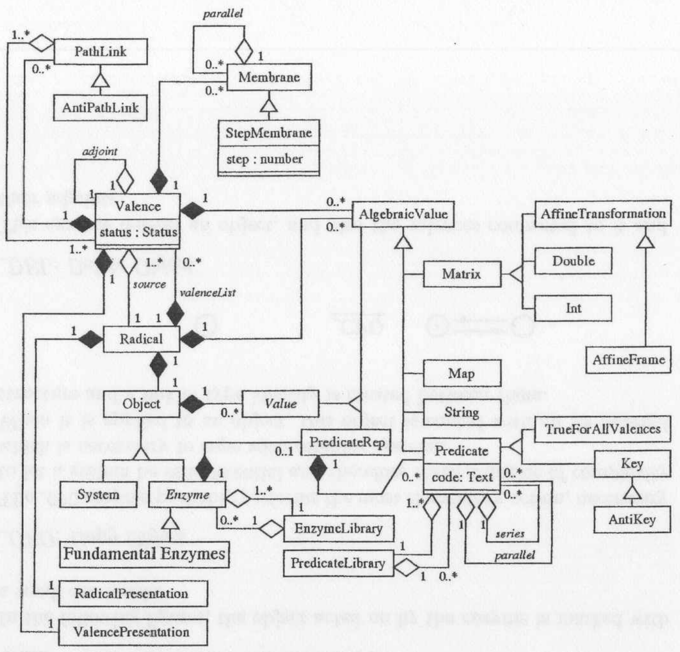


Figure 4.5: A more detailed class diagram of the core classes. The presentation classes are shown in 4.7. A class diagram of the language interpreter is shown in 4.8. The complete list of all classes can be found in appendix A.

a matching key, enzymes may pass a membrane. The classes **Key** and **AntiKey** are derived from **Predicate** to let enzymes verify keys as well as other predicates before they act (see the aFRK enzyme for further explaination).
Special enzymes can push objects inside or outside of membranes, by reassigning membranes. This way, a subsystem can grow or shrink.

4.2.1 Predicates

Predicates can provide conditions on an enzymes parameters to be fulfilled for the action to take place. Depending on the type of enzyme, predicates can act in two ways:

- A condition has to be fulfilled by the *radical, valence* pair for the enzyme to act.
- Some enzymes (like _AMR, _VML, _RMV) involve all the radicals valences in their action, (i.e. they have an implicit *forAll* quantor). In this case, the predicate restricts the action to those valences that fulfill the condition.

Since mechanisms can be composed to new enzymes, also called **enzyme chains**, an enzyme of the first type can be wrapped (with or without a predicate) inside an enzyme chain. This enzyme chain is an enzyme of the first type and can again be combined with a predicate.
As an example: `loop = _AMR{?iLP}` performs the _AMR action only with looped valences as parameters. The resulting chain (`loop`) can be combined with another predicate like ?hFK: The enzyme only acts if the radical is the central part of a fork structure:

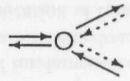


Table of Predicates

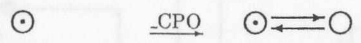
`?FLS, ?TRU:`	**alwaysFalse, alwaysTrue** - These predicates are mainly used for test purposes or to provide defaultvalues.
`?iLK, ?nLK:`	**isLink, isNotLink** - `?iLK` gives `TRUE` if a valence is fundamental and `FALSE` otherwise.
`?iID, ?nLK:`	**isIdentity, isNotIdentity** - Gives `TRUE` if a valence is of type `IDENTITY`. (This is the case for valences created in a copy process).
`?iAD, ?nAD:`	**isWithFundamentalAdjoint, isNotWithFundamentalAdjoint** - Checks whether a valences adjoint is fundamental or virtual
`?iLP, ?nLP:`	**isLoop, isNotLoop** - Checks whether a valences source and target are identical.
`?iSA, ?nSA:`	**isSelfAdjoint, isNotSelfAdjoint** - For loops, checks whether a valences adjoint is the valence itself.
`?iSP, ?nSP:`	**isSpike, isNotSpike** - This is `TRUE`, if a valences adjoint is the only valence pointing to that valences source.
`?hFK, ?nFK:`	**hasFork, hasNoFork** - Checks whether the radical is inside a fork structure.

4.2.2 The Fundamental Enzymes

In the following figures, the object acted on by the enzyme is marked with a small dot inside.

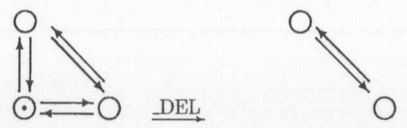
_CPO: Copy Object

The `_CPO` enzyme probably performs the most elementary action, neccessary to let a system be selfreferential and therefore reach a degree of complexity which is neccessary to cope with complex systems.
When it is applied to an object, this object is copied with all its internal structure and a link of type identity is created between them.



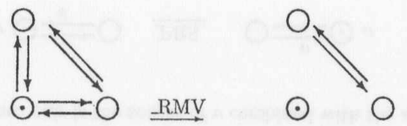
_DEL: Delete Object

This enzyme deletes an object, and also the valences connected to it and their adjoints.



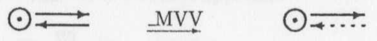
_RMV, _RTV: Remove Valences

The _RMV enzyme removes all valences (and their adjoint valences) pointing to the object, it is applied to. To remove only one valence, the enzyme _RTV (*remove this valence* can be used. It removes the parameter valence and its adjoint.



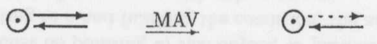
_MVV: Make Valence Virtual

Sometimes it becomes desirable to turn fundamental valences into virtual valences, in order to hide a link from mechanisms that move through a system (like shockwaves). Virtual valences are used by the *SplitFork* and by lattice growth enzymes (see below).
The _MVV enzyme needs an object and a valence as parameters. This valence gets a virtual status.



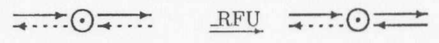
_MAV: Make Adjoint Virtual

The _MAV works analogous to the _MVV enzyme. It needs an object and a valence as parameters. This valences adjoint becomes virtual.



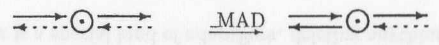
_RFU: Render Fundamental

If there are any virtual valences pointing to the object, the _RFU enzyme is applied to, they become fundamental.



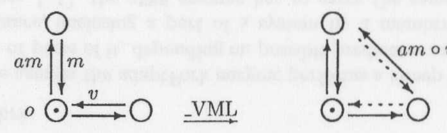
_MAD: Make Adjoints Fundamental

This enzyme creates fundamental adjoints to all valences pointing to the object, _MAD is applied to, which have no fundamental adjoint.

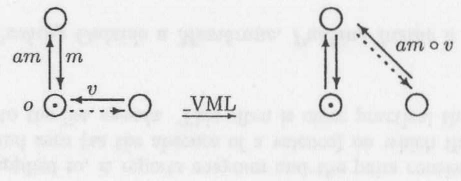


_VML: Compose Valences by Performing a Left Multiplication

The _VML enzyme takes an object o and a valence m, which must be pointing to this object, as parameters. All other valences v, pointing to o and fulfilling the condition, are composed with the adjoint am of m to $am \circ v$, where their values become $am * v$, and their new target changes from o to the target of am.

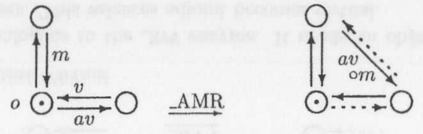


If the adjoint valence of v is virtual, two virtual valences would remain. As argued above, only links and arrows whose adjoints are links shall be represented. In such a case, the _VML enzyme acts in the following way:

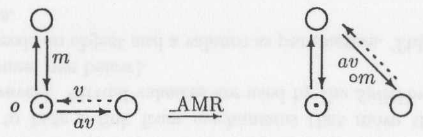


_AMR: Compose Adjoints of Valences by Performing a Right Multiplication

Analogous to the _VML enzyme, also the _AMR enzyme takes an object o and a valence m, which must be pointing to this object, as parameters. For all valences $v \neq m$, pointing to o and fulfilling the condition, the adjoints av of v are composed with m to form $av \circ m$ with values $av * m$.

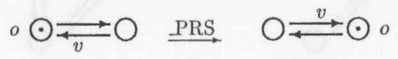


When v is virtual, the _AMR enzyme acts in this way:



_PRS: Present Source

The _PRS enzyme needs an object o an a valence v as parameters. The returned object-valence pair is the source of v combined with the adjoint of v.



_Rag: Report to Agenda

This enzyme reports enzymes and the object,valence pairs on which they are supposed to act to a list *agenda*. With this mechanism, sweeps through the whole system can be performed[4]

[4]While the agenda serves to implement the software on a single processor machine, it is thinkable to extend this concept by enzyme management to be used on multiprocessor machines.

_Sag: Report Sources to Agenda

For all valences v fulfilling the condition and pointing to the object o, the _Sag enzyme is applied to, it reports enzymes and the pairs consisting of the source of v and zero (as the absence of a valence) on which they are supposed to act to the list *agenda*. This often is more practical than the _Rag enzyme.

_POM, _PIM: Pushing Outside a Membrane, Pushing Inside a Membrane

These enzymes are used to rearrange membranes within a system. To create subsystems or to let them grow, shrink, or die, membranes can be rearranged to push an object, given as a parameter, inside or outside a membrane.

aFRK: Adapt Fork

Making use of the agenda the adaptFork enzyme performs a sweep through the whole system or parts of it, depending on possible predicates and membranes. For instance, enclosing a part of a system by a membrane constructed with a key `{.1}`, the aFRK enzyme has to carry the same key to sweep through the whole system (see figure 4.6). In order to block the enzyme only at a membrane with a certain key `{.x}`, an *AntiKey* `{-.x}` can be used.
Further enzymes, appended sequencially to the adaptFork, act on each object it passes.

aDEL: Delete Fork

The deleteFork is a special kind of adaptFork, deleting anything it passes.

_DEC: Decode Strings to Build Enzymes

When invoked and connected to an object of type string, this Enzyme reads out that string and produces an enzyme with that coding. This may be used to code enzymatic programs within the system.

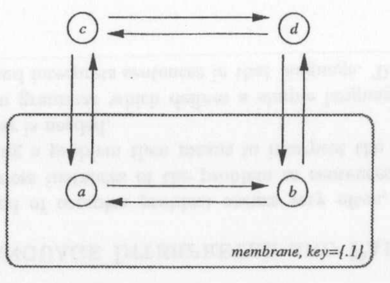


Figure 4.6: A system consisting of four objects a, b, c and d connected to each other. `_PIM{.1}` applied to a and b creates a membrane around a and b with the key `{.1}`, defining a subsystem. `aFKR{.1}_OUT` applied to any of the objects sweeps through the whole system and outputs each object it passes, `aFKR{.2}_OUT` applied to a or b only sweeps through the subsystem, but applied to c or d, it sweeps through the whole system again.

4.3 THE PRESENTATION

Once a simulation is invoked, there are many possibilities to present the obtained information. Some of the questions are: Shall the user interface be bidirectional? (i.e. shall the information just be presented, or should the user be allowed to influence the system via the presentation interface?) Is it necessary to interact with the simulation at runtime? How strong should elements of the core library be related to the elements of the presentation? Aiming for flexibility, the first two questions have to be answered with *yes*. The question of how strong the relationship should be is rather crucial. Should the elements be related via...

- Inheritance: The presentation *is* the core element,
- Strong ownership: The core element *has* a presentation,
- Weak ownership: The core element *has* a presentation with an existence of its own,
- An indirect control mechanism ?

Looking for an apropriate answer involves to see if any design patterns favouring one or another strategy. Inheritance, as an attempt to keep the

design small, led to an extremely inflexible structure: Switching between presentations was very complicated this way, since objects had to be destroyed and recreated whereby certain information had to be remembered somehow. Further more, changes made in the core classes had to be made as well in the derived presentation classes. All this made it necessary to choose a different design.

So far strong ownership seems to be the best solution. The *abstract factory pattern* [4] allows a clean separation of the core elements and their presentations. Since a presentation requires the existence of the element it presents, strong ownership is favoured over weak ownership. An indirect control mechanism might be even more flexible, but performance will as well decrease. Still this design can easily be extended to run different presentations.

More than one possible presentations can be realized. The decision which one to use is done at only one point in the program code: One of many *concrete factories* is created, which takes care of creating any other instances in context with the presentation, see figure 4.7.

The elements and their presentations are assigned to each other in both ways. Since a core element without a presentation does make sense, e.g. to keep the over all presentation clean, a presentation only weakly owns its core element. Both directions are necessary, allowing the user to manipulate the system via the user interface.

Two presentations are implemented so far. One is textual, based on the command line interface, the other one is a full graphical presentation, shown in appendix F.

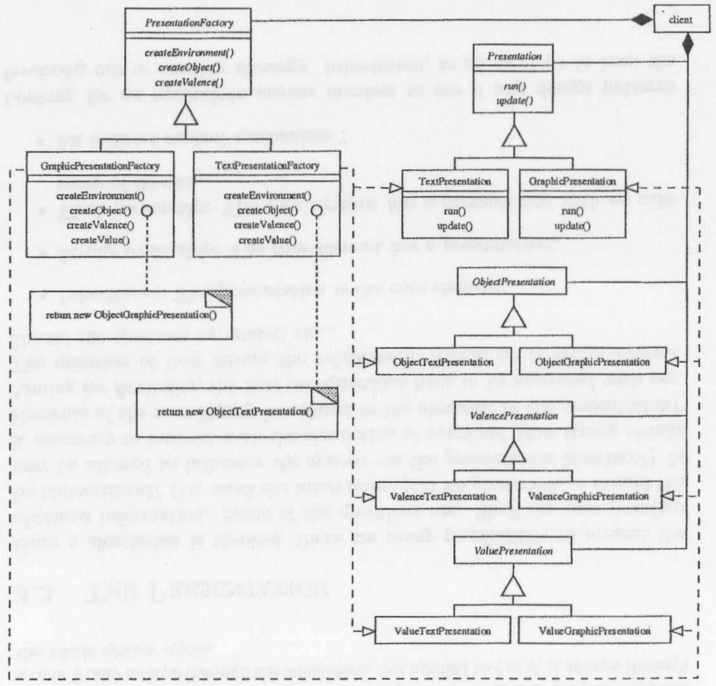


Figure 4.7: Class hierarchy of the SyCL presentation classes. Details of core classes are hidden inside the box **client** which is *not* a class. The *abstract factory pattern* is used here.

4.4 The Language Interpreter and Parser

If a particular kind of complex problem occurs very often, it might be worthwhile to express instances of the problem as sentences in a simple language[4]. Solving a problem then means to interpret the sentence, for which an *interpreter* is needed.
Based on a certain grammar which defines a simple language, the interpreter represents and interprets sentences in that language. To be efficient,

character strings which make up the sentences are transformed into trees of objects, which are easy to handle by the interpreter. This transformation is done by a software called *parser*.
Very frequent applications are pattern matching problems, in which the input string is checked against *abstract syntax trees*, which represent the grammar.

Consider an example grammar defining regular expressions:

```
expression ::= literal | alternation | sequence | repetition |
                    '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::=  'a' | 'b' | ... | 'z' | { 'a' | 'b' | ... | 'z' }*
```

Each line represents a rule, assigning the right hand side to the symbol on the left. The elements '|', '&' and '*' mean logical 'or', 'and' and repetition respectively (as declared selfreferentially). Characters enclosed in single quotes are uninterpreted.
The symbol `expression` is called the start symbol and `literal` is called terminal symbol - the smallest unit - used to define simple words. Implementing a grammar like this, a class is defined for each rule. Symbols on the right hand side of the rules are instance variables of the classes. Five classes are needed for this grammar: An abstract class **RegularExpression** and four subclasses **AlternationExpression, SequenceExpression, Repetition-Expression and LiteralExpression.** Note that all but the last contain subexpressions, which allows recursion.
Different grammars concentrate on different aspects of the problems under consideration. For system theoretic problems, a less complex grammar is chosen that proved to be very powerful:

```
element ::= list | atom
list    ::= '(' element* ')'
quote   ::= 'quote' atom
atom    ::= 'a' | 'b' | ... | 'z' | '+' | '-' | '*' | '/' | '$' |
            { 'a' | 'b' | ... | 'z' }*
```

The 'quote' is to be understood as a single quote. It is not implemented as a class of its own, but instead as a property of the class **Atom**, meaning an atom can be quoted or not. This becomes interesting when elements are

interpreted: Quoted elements remain uninterpreted.
In object oriented softwaredesign, a well known interpreter pattern (suggested in ref.[4]) has shown to work fine in most cases. To implement a parser and interpreter for the SyCL program package, this pattern is used in combination with a composite pattern [4] and extended by one more abstraction in the way shown in figure 4.8.

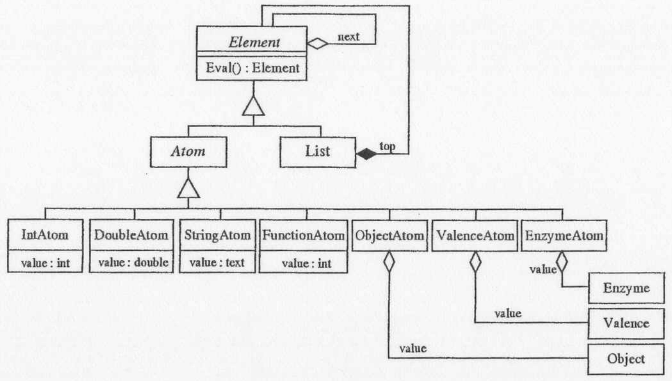


Figure 4.8: A class diagram for the language interpreter and parser.

The **Element** class is abstract. It provides polymorphism by assuring that evey object is of the type **Element**. An element can thus either be an object of type **List**, which consists of further elements, or an object of type **Atom**. The class **Atom** is abstract again. The following concrete classes are derived from it.

- **IntAtom**: Holds an integer value in strong ownership.
- **DoubleAtom**: Holds a floating point value in strong ownership.
- **StringAtom**: Holds a character string value in strong ownership.
- **FunctionAtom**: Holds an integer value, which represents a function, in strong ownership.

- **ObjectAtom**: Holds a SyCL object in weak ownership.
- **ValenceAtom**: Holds a SyCL valence in weak ownership.
- **EnzymeAtom**: Holds a SyCL enzyme in weak ownership.

Note that a *list* has a strong ownership relation to its elements (The elements are destroyed when the list is destroyed). Syntax trees generated with this structure are very similar to those used by list processing languages like LISP (see appendix C). A sample tree for an abstract SyCL expression

```
(function1 (function2 (enzyme1 object1 valence1) number2)
           (function3 number3 number4)
           number1)
```

is shown in figure 4.9.

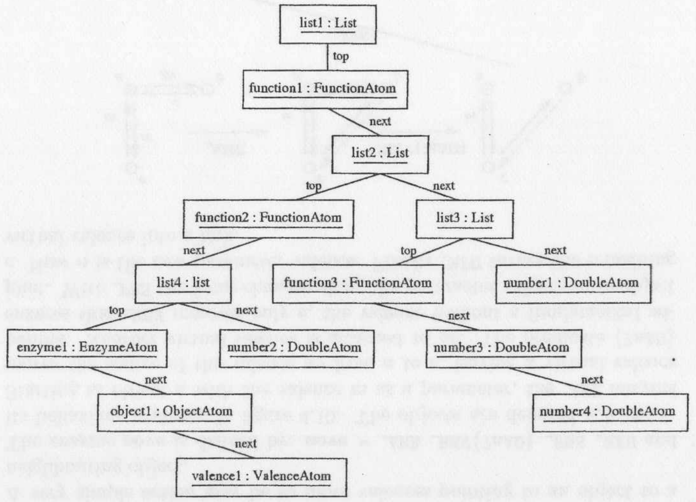


Figure 4.9: An object diagram for a sample SyCL expression.

A list can consist of elements of any kind. To evaluate a list, the first element has to be a function, or an enzyme, where the following elements serve

as parameters, which have to be evaluable unless they are quoted (i.e. they start with a quote). A complete list of implemented functions can be found in appendix D.
An enzyme acting on a system can be seen as the system theoretical equivalent to a function acting on a list in a list processor. Thus the interpreter implemented within this project does process both, lists and systems. The class hierarchy just introduced takes care of the list processing, while the SyCL classes manage the system processing. The essential idea about adding a list processor to SyCL is to provide a powerful tool to setup the system theoretical environment, invoke simulations and evaluate the results.
For convenience, variables can be defined. They are accessable to both the interpreter and to SyCL itself.

4.5 Enzymatic Programming

Given this environment, programming means setting up a system and composing enzymes and predicates to act on it. The following examples shall illustrate this idea.

4.5.1 `move`: Dynamic Action

A very simple action may be to move valences pointing to an object to a neighbouring object.
The enzyme `move` is defined by: `move = _AMR _RMV{?nAD} _PRS _RFU` and its behaviour is shown in figure 4.10. The objects are denoted a, b and c. Starting at object a with the valence m as a parameter, the `_AMR` enzyme moves the source of the valence av from a to c, leaving a virtual valence behind. Another virtual valence is attached to av. The predicate `{?nAD}` ensures that `_RMV` removes only v, the valence without a fundamental adjoint. With `_PRS` the focus changes along the parameter valence m to object c. Now n is the new parameter valence. Finally `_RFU` turnes the remaining virtual valence into a link.

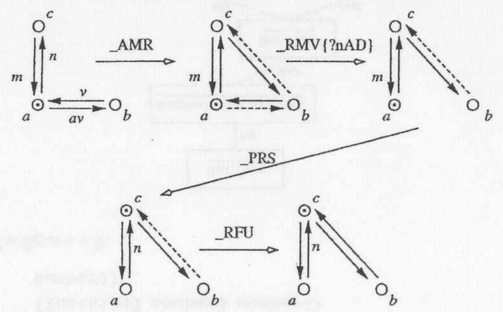
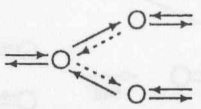


Figure 4.10: The behaviour of the `move` enzyme. The object under focus is marked with a small dot.

A Sweep Through a System

A very elementary operation on an arbitrary system is to create a copy. The first approach is to perform a sweep through the system, making use of the *agenda*, and to copy each element during this sweep.
It is interesting to see how the directions in which to proceed can be declared: As in a shockwave, the links that have just been passed shall not be passed in backward direction immediately afterwards. Rendering these valences virtual reduces the neighbourhood (defined via fundamental valences) to valences the sweep has not passed. Thus propagation is as simple as proceeding into the neighbourhood.
These virtual valences with fundamental adjoints delimit elements from the neighbourhood and can therefore be understood as the prototypes of membranes.
During a copy process, the structure occuring in these situations



is called a **fork** and functions as a valve, keeping the sweep from moving backwards.

4.5.2 sFRK: The SplitFork - Replication of a System

The SplitFork enzyme responsible for a reproduction of a simple system (without loops) is composed in the following manner (see figure 4.11): sFRK = _CPO _AMR{?nLK} _VML{?iAD} _MAD _PRS _RFU _RMV{?iID} _Sag{?nAD}

It performs one step in the copying process of a system: The object, the first enzyme is applied to is colored in this diagram. The _CPO enzyme copies it, creates an identity valence and uses this as the valence-parameter, being passed to the _AMR enzyme. This moves only the virtual valences to the copied object, because the predicate {?nLK} has been specified. The action of the following enzymes can be seen in figure 4.11. Finally the _Sag enzyme reports the neighbourhood to the agenda. The remaining virtual valences are rendered fundamental in the next step.

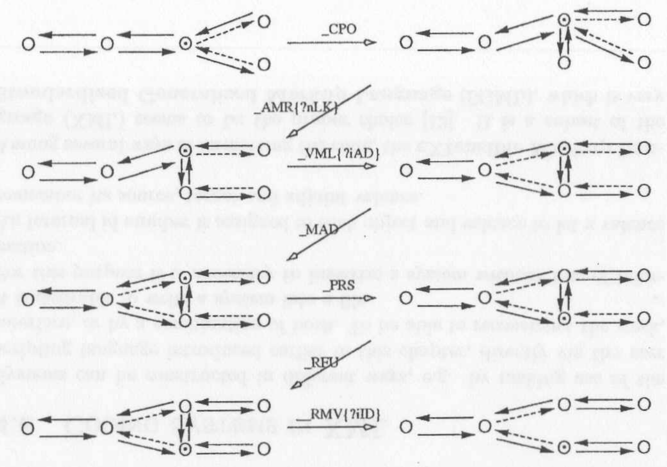


Figure 4.11: One step within a sweep of the SplitFork enzyme.

4.5.3 sFKL: The Loop Proof SplitFork

A closer look at the sFRK enzyme reveals that it causes trouble when there are loops among the systems valences. To handle this problem, another (loop-proof) enzyme can be found (see figure 4.12), it is composed via: sFRK = _CPO _VML{?iLP&&?iAD} _RFU{?iLP} _AMR{?nLK} _VML{?iAD&&?nLP} _MAD _PRS _RFU _RMV{?iID} _Sag{?nAD}.

In contrast to the _sFRK enzyme loops get a special treatment (_VML{?iLP && ?iAD} _RFU{?iLP}) before the _AMR enzyme acts, and the _VML enzyme gets restricted to non loops.
Without this treatment loops within the system lead to loops in time: The same objects are reported to the agenda again and again, the copy process never ends.
Since virtuality is used to determine the sweep propagation, only systems without virtual valences are replicated correctly. To get rid of this restriction, another SplitFork can be implemented, which determines its propagation on the base of membranes.

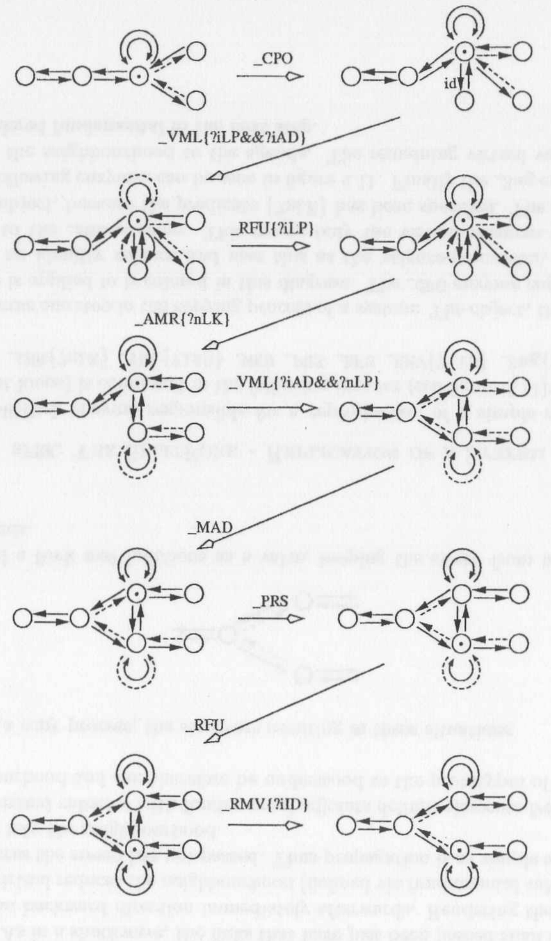


Figure 4.12: The loop-proof SplitFork enzyme sFKL

4.5.4 `latt`: Lattice Growth

A lattice shall grow by connecting a copy to itself: Given a (e.g. linear) system, copies are made and connected to each other sequencially. Afterwards, the same procedure can be applied to the new (2-dimensional) system to build a 3-dimensional one. This way, a lattice can be constructed, starting from an arbitrary small system.
The enzyme `latt = _CPO _AMR{?nLK} _VML{?iAD} _MAD _PRS _RFU _MVV _Sag{?nAD} _RFU` does make a copy while attaching each duplicated object to its original via an unidirectional link. This can be applied repeatedly to let the system grow in one dimension. With a system sweep, adding adjoints to the unidirectional links, this state is fixed and another application of the `latt` enzyme lets the system grow in a higher dimension.

4.5.5 `cell`: Cell Replication

Cellreplication can be achieved by realization of the following idea: A cell A gets triggered to replicate, for instance by a certain value of a gradient to a neighbouring cell B. A copy C of A is made and placed between A and B in order to smoothen the gradient. Therefore the fundamental links between A and B are replaced by new fundamental links between B and C. Furthermore C obtains new fundamental links to all neighbours, A and B are both connected to (see figure 4.13 and 4.14).

4.6 Coding Systems in XML

Systems can be constructed in different ways, e.g. by making use of the scripting language introduced earlier in this chapter, directly via the user interface, or by a combination of both. To be able to reconstruct the work, it is desirable to write a system into a file.
For this purpose it is necessary to linearize a system without loss of information.
An internal id number is assigned to each object and valence to let a valence remember its source, target and adjoint valence.

Among several ways of formatting the code, the **eXtensible Markup Language** (XML) seems to be the proper choice [12]. It is a subset of the **Standardized Generalized Markup Language** (SGML), which is very

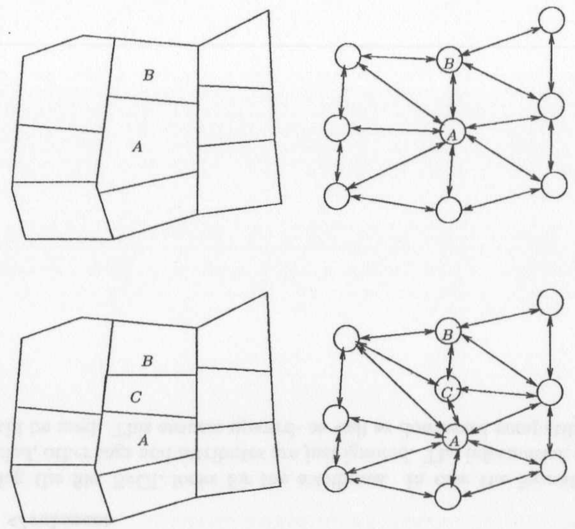


Figure 4.13: Cellreplication.

powerful, but too extensive to be used in this case.
One of the benefits of XML is that it allows upward as well as downward compatibility easily to be implemented.

Coding in XML

In XML, information is kept inside *tags* declared in a *document type definition* (dtd). They are notated in brackets, enclosing the tag name as well as attributes. Besides attributes, other tags may also be included. For this purpose, to the just defined *start tag* there also exists an *end tag* notated by a slash infront of the name, enclosed in brackets.

```
<tag attribute="something">
...
</tag>
```

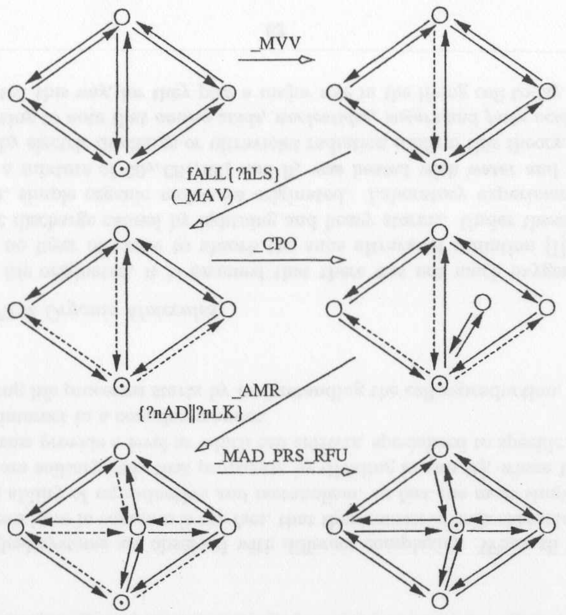


Figure 4.14: Cellreplication in a system theoretic diagram.

As a special case, a tag containing no further tags, does not need an *end tag*. It is notated by an extra slash in front of the close bracket.

```
<tag attribute="something" />
```

Coding Systems

The SyCL dtd (listed in appendix B) defines the following tags and their possible attributes.

- system: Anything else is enclosed inside the system tags.

- **object**: An object tag carries its id number and a position as attributes. *Value* tags may be enclosed.
- **valence**: A valence tag carries its id number and the id numbers of its source and target objects and of its adjoint valence as attributes. *Value* tags may be enclosed.
- **value**: A value tag has no end tag. It carries the value type, the dimenstion (in case of a matrix) and the content as attributes.

A sample system can be found in appendix B.

Writing a system into a file, first all objects and their values are written.

```
<object ID="7" POS="{-139,31,0}" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</object>
```

In the second step, all valences are written. This way the system can easily be reconstructed when reading from the file.

```
<valence ID="4" SOURCE="3" TARGET="2" ADJOINT="5" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</valence>
```

Reading the file, SyCL looks for the attributes. In case the format gets extended, other tags and attributes are just ignored. The information given can still be used. This ensures upward- as well as downward compatibility.

Chapter 5

Biological Systems

To understand the methods used to generate artificial life, which will be discussed in the next chapter, it is important to study the mechanisms nature uses. This chapter shall give a short description of the metabolism and reproduction mechanisms in simple biological systems (see [1]).

Biological systems are observed with different complexity. What all living creatures have in common is the fact, that they consist of cells characterized by the ability of reproduction and metabolism. In fact the most simple life forms are solitary cells that propagate by dividing in two [1], where higher organisms provide a level at which cell clusters, specialized to specific functions, interact in a complex manner.
Studying life processes starts by understanding the cell reproduction.

The First Organic Molecules

When life originated, it is assumed that there was not much oxygene on earth, no layer of ozone to absorb the suns ultraviolet radiation [1], but electric discharge caused by lightning and heavy storms. Under these conditions, simple organic molecules originated. Laboratory experiments in which a mixture of CO_2, CH_4, NH_3 and H_2 was heated with water and energyzed by electric discharge or ultraviolet radiation confirm this theory. It is interesting to note that *amino acids, nucleotides, sugars* and *fatty acids* are generated this way, for they play a major role in the living cell today. The

simple molecules under consideration are *amino acids* and *nucleotides*.
There are four types of nucleotides, *uracil, adenine, cytosine* and *guanine*, abbreviated U, A, C and G. They can associate by phosphodiester bond to form large polymeres, the *ribonucleic acid* and *desoxyribonucleic acid* - RNA and DNA - molecules. The amino acids on the other hand can join together by peptid bond to form large polymeres: A set of twenty amino acids constitutes the universal building blocks of the proteins.

Nucleotides: A, U, G, C	associate to	polynucleotides: RNA, DNA
20 amino acids	associate to	polypeptides: Proteins

Catalytic Activity

With the loss of water, spontaneous polymerization occurs. The resulting molecules have random lengths and sequences. Furthermore, preferential binding occurs between pairs of nucleotides: C with G and U with A. This is a very important point for the origin of life, for it allows a simple reproduction of polynucleotides: Once a certain polynucleotide has formed and spontaneous polymerization occurs, the existing polynucleotide binds other nucleotides and functions thus as a template for new polynucleotides. This mechanism applied twice results in the original polynucleotide: A copy has been established.
For templating mechanisms to be efficient, catalytic mechanisms are required. These catalytic functions are provided by highly specialized proteins called *enzymes*, but also RNA molecules have the potential to catalyse specific reactions, by folding up to form complex surfaces.
Folding of polynucleotides rises from the preferential bindings between the nucleotide pairs. Two parts of the same polynucleotide bind, when they are complementary in their sequences, (see figure 5.1). The folding of polymeres

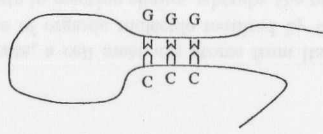


Figure 5.1: A RNA molecule folds up to a characteristic spatial geometry when two parts of the same molecule bind to each other.

influences their stability as well as their ability of replication. It happens

that errors occur in the replication process and imperfect copies of the original will be propagated. This way, replicating systems of RNA molecules undergo a form of natural selection. The two essential characteristics of RNA molecules therefore are[1]:

- Informational: RNA carries information encoded in its nucleotide sequence, that is passed on with replication.
- Functional: RNA is folded in a unique way, that determines how it interacts with other molecules and how it responds to different conditions.

The informational characteristic is analogous to the *genotype* (the hereditary information), the functional characteristic is analogous to the *phenotype* (the geneic information of which natural selection operates) of an organism.

5.1 The Cells Structure

The development of an outer membrane procures an evolutionary advantage. For a certain RNA to facilitate its own reproduction, it is necessary for the proteins generated under control of that RNA to remain in the RNAs neighbourhood. Furthermore, they could in a different environment facilitate other RNAs reproduction. This is assumed to be the origin of the cell[1].
Compartment is easily achieved by simple *amphipathic* molecules, consisting of one hydrophobic part and one hydrophillic part. Placed in water, these molecules spontaneously aggregate to form bilayers, creating small closed vesicles whose aqueous contents are isolated from the external medium.
While RNA is supposed to have come first in evolution, DNA has taken over the primary genetic function at some point in time, after efficient protein synthesis had been evolved. Proteins became the major catalysts, while RNA remained as the major link connecting the two. Certain chemical differences between RNA and DNA fit them to perform specialized functions. DNA is more stable than RNA and therefore seems better suited to permanently store genetic information. This is partly due to a lack of a sugar hydroxyl group, but also because DNA, unlike RNA, exists principally in a double stranded form. This allows DNA molecules to be easily replicated, as well as repaired, since the two complementary strands contain the genetic information twice.

The single stranded RNA molecules control the protein synthesis in two ways, as coding RNA molecules (*messenger* RNAs) and as RNA catalysts (*ribosomal* and other non-messenger RNAs).
In higher organisms, most of the cells DNA molecules are arranged inside a nucleus, which is separated from the other cell material by a double layer of membrane. Cells with a nucleus are called *eucaryotic*, whereas those without a nucleus are called *procaryotic*.

Metabolism

In order to replicate, a cell must use atoms from its environment to synthesize every type of organic molecule required by the cell. Hundreds of enzymes collaborate in reaction chains, whereby the product of one reaction is the substrate for the next, and catalyze this process. These enzymatic chains are called *metabolic pathways*.
An interesting aspect is the gain of energy through a sequence of reactions known as *glycolysis*, the degeneration of glucose in the absence of oxygen (i.e. *anaerobical* degeneration) on one hand, and *respiration*, the (*aerobic*) oxidation of food molecules on the otherhand. The energy release through respiration (complete degeneration of glucose to CO_2 and H_2O) is much more efficient than through anaerobic glycolysis, where glucose can be broken down only to lactic acid or ethanol. Both processes drive the formation of *adenosine triphosphate*, *ATP*, which is used by all cells as a source of energy.
For large cells the ratio of surface to volume is too small to achive the complete metabolism by pushing the material through the pumps and channels of the cell membrane. To manage that problem, large parts of the cell membrane are carried directly into the cell by a process called *endocytosis*, and the reverse process, *exocytoses*, where new cell membrane is built from the membrane that is enclosed inside the cell.

The Cell Organelles

The cells boundary is its plasma membrane, which encloses the nucleus (in eucaryotic cells) and the *cytoplasm*, where most of the cells metabolic reactions occur. About half of the total cell volume is occupied by further compartments, the *organelles*, which are listed below for an eucaryotic animal cell. The remaining space, which includes anything else, but the membrane bounded organelles, is referred to as *cytosol*. A picture of an eucaryotic animal cell is show in figure 5.2.

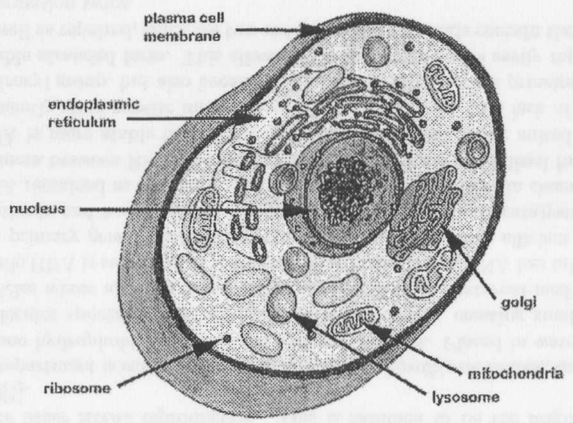


Figure 5.2: Picture of an eucaryotic animal cell.

- Plasma Membrane: The plasma membrane is a continous sheet of phospholipid molecules in which various proteins are embedded. Some of these serve as pumps and channels for transporting specific molecules into and out of the cell.
- Endoplasmic Reticulum: Flattened sheets, sacs, and tubes of membrane extends throughout the cytoplasm of eucaryotic cells, enclosing a large intracellular space. The ER membrane is in structural continuity with the outer membrane of the nuclear envelope and it specializes in the synthesis and transport of lipids, membrane proteins as well as other material destined for export from the cell. The *rough ER* is studded on its outer face with ribosomes engaged in protein synthesis. The *smooth ER* lacks attached ribosomes. A major function is in lipid metabolism.

- Golgi Apparatus: A system of stacked, membrane bounded, flattened sacs involved in modifying, sorting and packaging macromolecules for secretion or for delivery to other organelles. Around the Golgi Apparatus are numerous small membrane bounded vesicles, which are thought to carry material between the Golgi Apparatus and different compartments of the cell.
- Lysosomes: Membrane bounded vesicles that contain hydrolytic enzymes involved in intracellular digestions. The membrane prevents the lysosomes from attacking proteins and nucleic acids elsewhere in the cell.
- Peroxisomes: Membrane bounded vesicles that contain oxidative enzymes that generate and destroy hydrogen peroxide.
- Nucleus: Separated from the cytoplasm by an envelope consisting of two membranes. All of the chromosomal DNA is held in the nucleus, packaged into chromatin fibers by its association with an equal mass of histone proteins. The nuclear contents communicate with the cytosol by means of openings in the nuclear envelope, called nuclear pores.
- Centriole: A small cylindrical organelle. Centrioles exist in pairs, with two centrioles at right angles to each other.
- Cytoskeleton: In the cytosol, arrays of protein filaments form networks that give the cell its shape and provide a basis for its movements. The main kinds of cytoskeletal filaments are *microtubules, actin filaments* and *intermediate filaments,*
- Mitochondria: About the size of bacteria, mitochondria are the power plants of all eucaryotic cells, harnessing energy obtained by combining oxygen with food molecules to generate ATP.
  They alone are responsible for respiration in the eucaryotic cell. The H^+ gradient, required for ATP production, does not occur at the plasma membranes (as it does for procaryotic cells), but is totally deligated to the mitochondria. This results in a higher flexibility for the plasma membrane, which is therefore capable of controlled changes in the ion permeability for cell signaling purposes.
  Since mitochondria show many similarities to aerobic bacteria (they both contain DNA, generate proteins, reproduce by dividing in two, make use of respiration and are similar in size and shape), it is thought that eucaryotic cells descended from primitive anaerobic organisms, that survived the increasing amount of oxygen by a symbiosis with aerobic bacteria.

Plant cells differ from animal cells generally by the lack of centrioles and addition of three further organelles: The ***chloroplasts*** consist of an internal membrane system and chlorophyll, which gives them the capability to drive photosynthesis. *Vacuoles* are large single membrane bounded vesicles occupying up to 90% of the cell volume, responsible for intracellular digestions. The *cell wall* is composed of tough fibrils of cellusose laid down in a marix of other polysaccharides.

5.2 Cell Replication

Essential for reproduction of any simple or complex cellular organism is the cell replication. The way cells reproduce is to duplicate their contents and then dividing in two. It can be viewed as the smalles entity, that encloses all that is necessary to reproduce itself.

Interphase, Mitosis, Meiosis

The life cycle of a cell can be divided into two parts, the interphase and the cell division phase. During the **interphase** the cells genetic material is active, its nucleus is separated from the plasma by its membrane. The cell grows to twice its mass. With the end of the interphase, the chromosomes within the nucleus are duplicated but remain connected.
During the **cell division phase** or **Mitosis**, the necleus is duplicated and cell division takes places. It is again divided into subphases, briefly described here:

- **Prophase**: The chromatin slowly condenses into well defined chromosomes, each consisting of two *sister chromatids*, created at the end of the preceding interphase. Toward the end of the prophase the *mitotic splindle* begins to form. It assembles initially outside the nucleus.
- **Prometaphase**: The nuclear envelope disrupts, breaking down into small membrane vesicles. The spindle microtubules now enter the nuclear region while the splindle poles move to opposite parts inside the cell, keeping tension on the chromosomes.

- **Metaphase:** The chromosomes are eventually aligned in one plane halfway between the spindle poles. Each chromosome is still held in tension by the microtubules, which are attached to opposite poles of the spindle.
- **Anaphase:** Triggered by a specific signal, the microtubules separate the chromatids from another. They are slowly pulled toward the spindle poles.
- **Telophase:** Around each group of daughter chromosomes a new nuclear envelope is built and the condensed chromatin expands again.
- **Cytokinesis:** Between the two nuclei, cell membrane is built, leaving two separated daughter cells.

The regulation of this cycle is done by a protein known as *MPF* (maturation promotion factor), which again is regulated by a protein namens *cyclin*, see figure 5.3.

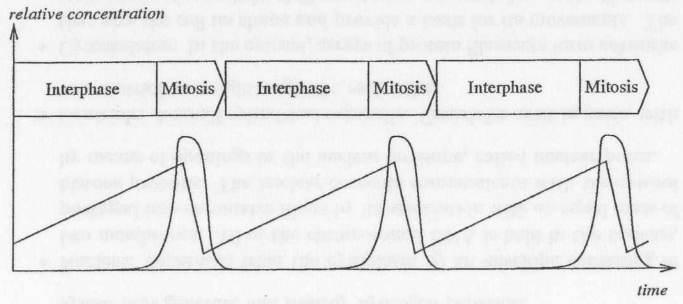


Figure 5.3: The correlated rise and fall in the levels of MPF and cyclin during a cell cycle.

These periodic behaviour in time functions as a natural timer to trigger cell replication.

Higher organisms that replicate sexually carry a double set of chromosomes, one set given by the father, another one by the mother. During sexual reproduction, this double set must be reduced to a single set of chromosomes, to prevent a duplication with each generation and thus infinite growth in

the long term.
This reduction of the double set of chromosomes is called **Meiosis**. Mitosis and Meiosis are triggered by the same molecules.

Recombination Mechanisms

Meiosis only occurs with sexual reproduction. Asexual reproduction is therefore simpler, but on the other hand less flexible in changes of the genetic material. Evolution takes place mostly by simple point mutations due to errors during the copying process of DNA or RNA. This does not provide many possibilities of meaningful changes.
Sexual reproduction in contrast involves many possibilities to rearrange the genetic material during the meiosis, most important *general recombination*: Genetic exchange takes place between homologous DNA sequences, usually located on two copies of the same chromosome. This ensures that compatible material is exchanged, manipulating genetic properties, not the overall structure of a chromosome. This *crossing over* increases the chance to survive in a changing environment.
DNA rearrangements occur not only during sexual reproduction. Besides general recombination, *site specific recombination* is another important mechanism: Specific sites on the DNA are recognized by certain enzymes, opening the molecule and inserting other DNA sequences.

These mechanisms allow a vast variety of changes during life and reproduction. It is a subtle task to balance between stability and flexibility in a changing environment.

Chapter 6

ARTIFICIAL LIFE

This chapter shall give a few examples from the area of artificial life.

6.1 JOHN VON NEUMANN

In context with artifical life, the most important person to mention is the one who invented this discipline, John von Neumann.
Von Neumann regarded life as a reconstructable linkage of events and interactions [11]. In his lecture *The General and Logical Theory of Automata*, he suggested that a deeper understanding of automata led to a deeper understanding of the mechanisms of life itself. He also came up with a concept of self replication. Information being the foundation of all life he was conviced that a system of any kind up from a certain barrier of complexity should be able to produce new systems even more complex than itself.
Von Neumann developed a concept for a self replicating automaton. Aside from electronic material (logical switches *and*, *or*, *not* and delay circuits), his machine consisted of a *manipulation unit* (a roboter device, that could obtain orders from the central processing unit, the CPU), a *cut unit* to disconnect elements on order of the CPU, a *connection unit*, a *sensor unit* that detects elements and transmits their information to the CPU, and a couple of *carrier elements* relevant for the structure of the whole automaton as well as the information memory. As a space for this machine to live in, he assumed a large reservoir in form of an infinite sea, filled with the same

elements, the machine is made of, randomly distributed.
This machine was able to reproduce itself, by executing commands from a construction plan that was encoded by certain carrier elements. After reconstructing the machine itself, it transmitted the construction plan. It is interesting to note that he had introduced this concept a few years before the DNA molecule was discovered.
One of the main disadvantages of his concept was that most of the units which had to be given in the reservoir, were far more complex than the architechture of the machine itself. So this could not explain how life originates from a reservoir of simple objects. Von Neumann disliked this too, so he decided to choose a different (simpler) basis for his automaton: Continuing the ideas from Stanislaw Ulam, he developed the idea of the cellular automaton. On this foundation he created an object that was able to replicate itself [11].

6.2 CELLULAR AUTOMATA MACHINES

Intuitively, a cellular automaton is a network of identical, uniformly interconnected objects, combined with a rule (the *local map*) which determines their dynamical behaviour in a local manner. To each object, or *cell*, there is associated a state variable, called the *cell state*, ranging over a finit set of values, the *alphabet*; and a cells *neighbourhood* is defined as the set of cells directly connected to it through the network.
With each time step (time and space are discrete!) the new state of each cell is determined depending on the (former) states of the cells neighbourhood. Thus, a cellular automatons laws are *local* and *uniform* [26]. In this respect, they reflect two fundamental laws of physics.
An assignment of states to all cells is called a *configuration*. Applying a local map to each cell results in a new configuration. Thereby a local map defines a transformation, called the *global map*, on the set of configurations.

A cellular automaton is invertible, if its global map is invertible. Invertibility is to be understood as conservation of information. This is an important point in context with dynamical systems, since it represents microscopic reversibility [26]. When trajectories meet in phase space, which holds for probably most cellular automata, it is of course not invertible.

Von Neumanns Selfreplicating Automaton

Von Neumann chose an alphabet consisting of 29 elements [11]. Starting with a horizonless grid, with each cell in an inactive state, an organism was introduced covering two hundred thousand cells. The details of this creature were represented by different states of individual cells.
The organism consists of parts shaped like a box and a tail, where the box contained suborganisms: A *factory* (arranging information from the environment according to instructions from other suborganisms), a *duplicator* (reading informational instructions and copying them) and a *computer* (functioning as a control apparatus).
These components took up only a quarter of the creature's total number of cells. The rest of the cells were in a single file line of 150000 cells, acting as a blueprint for the instructions to construct the entire organism (see figure 6.1).

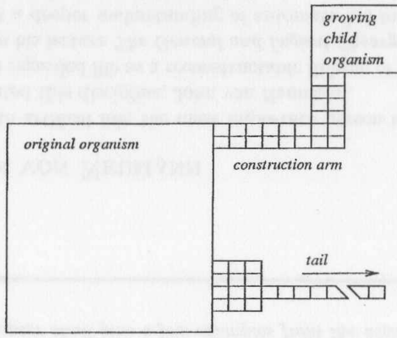


Figure 6.1: A schematic view of the selfreplicating automaton. The tail contains 150000 cells, coding the replication procedure.

Once this automaton was embedded in the grid, each cell, as an individual finite state machine, began to follow the rule that applied to it. The effect of these local behaviours caused a global behaviour to emerge: The self-reproducing structure interacted with neighbouring cells and changed some of their states. It transformed them into the materials - in terms of cell states - that made up the original organism. The tail of the organism contained instructions for the body of the creature. Eventually, by following rules of

transition (drawn up by Von Neumann), the organism made a duplicate of its main body; information was passed through an 'umbilical cord', from parent to child. The last step in the process was the duplication of the tail, and the detachment of the 'umbilical cord'. Two identical creatures, both capable of self-reproduction, were now on the grid [11].

Conways Game of Life

Initially, cellular automata were mainly used as toy models for phenomenology of dissipative processes [26], as biological organization, self-reproduction, chemical reactions and visual pattern processing.
John Conway was experimenting with cellular automata using an alphabet that consists only of two states: Zero and one, which he called *dead* and *alive*. Starting from statistically distributed configurations, most maps lead to either infinite growth or quick death of cells alive. Conway found a local map, that lead to an quasi-equilibrium, where statistically distributed configurations lead to a non-zero stable number of living cells, possibly changing periodically.

The *Game of Life's* neighbourhood consists of eight neighbouring cells, not including the cell itself. The local map determines the state of a cell depending only on the number of alive cells in the neighbourhood, neglecting their exact position. Starting with a symmetric distribution of alive cells on the grid, the system will always stay symmetric.
A cell can change its state in four ways: *Change to zero (death), change to one (birth), stay the same* and *invert the state.* Depending on the number of alive neighbours, the local map can be stated a table:

number of alive neighbours	state of the center cell
0	0
1	0
2	no change
3	1
4	0
5	0
6	0
7	0
8	0

These rules can be interpreted in the following manner: Three alive neighbours are needed for a new cell to be born, two are needed for a cell to stay alive. A cell with less neighbours will die, suffering from underpopulation. Any number greater than three leads to death as well in consequence of overpopulation.

Experimenting with this environment, a lot of objects can be found, even starting with random distributions. An object consisting of three alive cells in a row (vertically or horizontally) is called a *blinker*, since it changes from a horizontal to a vertical arrangement and vice versa.
A more complex structure is called a *glider* (see figure 6.2), changing in shape and position until after four steps in time, it rearranges to its original shape at a new position. These gliders move diagonally over the grid.

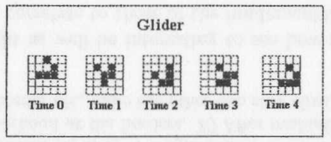


Figure 6.2: The *glider* moves diagonally with four steps in time.

Many objects retain their shape and position periodically. They are called *oscillators* (see figure 6.3).

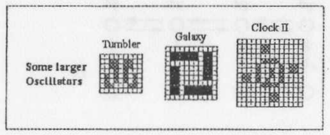


Figure 6.3: *Oscillators* always return to their original shape and position.

Extending Cellular Automata Machines

Object structures living on a grid of a cellular automaton are recognized as such by the observer. In context with the automaton, there are no objects other than cells. These cells are associated with specific positions on the

grid, they don't own their position as an attribute.

Thus the concept of a cellular automaton does not go well with the object oriented paradigm. These problems can be overcome by porting a cellular automaton to the system theoretical framework: Most essential is to construct the grid from objects. Neighbours can be defined in terms of fundamental arrows. This allows to manipulate the grid's geometry in any way: Such a grid can differ in density, introducing singularities like holes (missing objects) or accumulations (addidional objects), resulting in non constant neighbourhoods. Wormholes can be introduced by connecting geometrically separated objects by fundamental arrows. By adding loops, a cell can be its own neighbour.

This has been implemented with SyCL. A grid is created by a SyCL script. Each cell is implemented as an object connected to a position on the grid by a unary link (see figure 6.4), which carries the state of the cell as a value. For convenience this value is copied to the cell object.
Two enzymes are responsible for the dynamics:

- The `dist` enzyme does a sweep through the system multiplying the valences carrying the state values with all neighbourhood valences (see figure 6.4). It is composed from fundamental enzymes by
  `dist = aFRK fALL(_AMR{?nAD}){?iLK&&?iAD})`
- The `_CAM` enzyme is fundamental. It collects the values of the unary links, evaluates them according to the local map and thereby sets the state of a cell. To act on the whole system it has to be combined with the AdaptFork:
  `ccam = aFRK _CAM`

This way a fully functional cellular automaton is implemented, including the ability to change the geometry of the system.
The SyCL script can be found in appendix D, example 5.

Outlook: Multiscale Automata

It might be interesting to experiment with cellular automata on different scales as well. Blocking to a coarser lattice, the effective local map will not be well defined in most cases. This problem might be solved by introducing stochastical maps: The update of each cell takes place with a certain

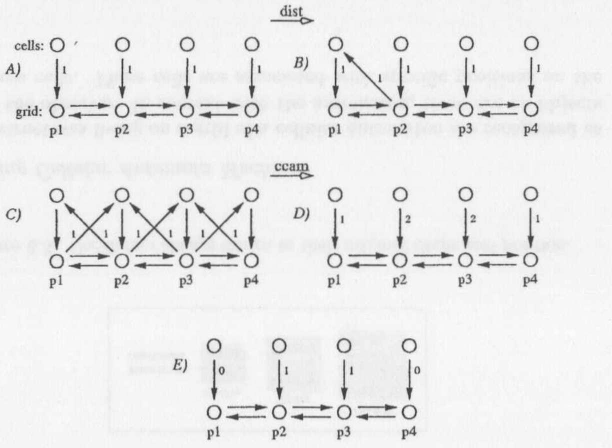


Figure 6.4: *A)* A view on a one dimensional grid with a neighbourhood consisting of two objects. Each cell has a state of one as an example. *B)* Action of the `dist` enzyme at position p1. *C)* Action of the `dist` enzyme on the whole system. *D)* The state values collected by the `ccam` enzyme before evaluating the local map. Note that the distribution (1, 2, 2, 1) is symmetric, but not homegenuous due to the differing neighbourhood at the borders. *E)* After evaluation of the local map, the two cells at the border die, while the other two stay alive.

propability. It might as well be interesting to see how the propabilities of the effective action correlate to those of the fundamental action.

6.3 The Primordial Soup

The Santa Fe Institute in New Mexico is well known for its research on artificial life. The *Primordial Soup* is a life simulation program, developed at the SFI [5].

An organism is represented by a piece of software, being executed over and over. These software loops live in a shared memory space, the *soup*, with the ability to self-reproduce. For time evolution, a multitasking interpreter is used. One instruction is executed from each organism in turn, so that the

organisms effectively run simultaneously.
In accordance with Darwinian evolution principles, the organisms mutate and evolve. Replication of an organism is implemented via copying its program code to another place in the memory space and starting a new process, executing at the start of the copied block.
Copying program code, the code of another organism may be overwritten. This provides a limited form of sexual reproduction: The organisms share genes by the mechanism of one partially overwriting another. Another source of mutation is given by a simulation of background radiation: Bits of the memory are complemented at random with a controlable rate.
The program may be started with one or more organisms seeding the soup. Alternatively, the system may be started *sterile*, with no organisms present. Spontaneous generation of self-reproducing organisms has been observed after runs as short as 15 minutes.

6.4 The Idea of an Enzymatic Game of Life

The SyCL environment provides the essential elements to simulate artificial life. It is essential to understand that by a clean separation of scales, one does not have to start implementing atoms, molecules and their properties. It suffices to start on the scale of interest, which is the scale on which evolution takes place.
A simple approach may be to create a universe of different kinds of enzymes. A SyCL script can be found in appendix E. Since some of the fundamental enzymes are very specific, it may be reasonable to start with a given set of enzymes.
They are randomly distributed and their relations as well.

To start the simulation, each enzyme acts on the source of one of one its valences with the adjoint valence as a parameter. It can be seen that the system grows. Unfortunately no technique is implemented so far to analyse the data for evolving structures. This is a point to be followed up in future.

Chapter 7

OUTLOOK

The perspective for the System Class Laboratory can be divided into hardware and software. On the hardwarelevel, a new processordesign is thinkable to be based on systemtheoretic manipulations. Since only very few operations are fundamental, it might push the concept of the *reduced instruction set computer* (RISC) even further, resulting in an enormous increase of performance.
On the softwarelevel, elaborate methods are needed to analyze the data. Making use of the software, more composite enzymes (or *macroenzymes*) can be defined, performing frequently occurring tasks.

As a special application, the enzymatic game of life can be continiued. A lattice of objects can be defined to represent the space. Associated to each object of the space lattice and connected via fundamental valences are

- the objects and enzymes that make up the organisms, and
- a system representing the material in the environment of that point in space.

To ensure conservation of material, the copy process would have to be exchanged by a process which takes the object to be replicated from the environment. In case of absence of a certain material, the process aborts. This way the natural struggle for bare resources can be implemented in the simulation.

Further more a general timer and a way to mutate the material has to be implemented. Mutation might be implemented via enzymes. One could start with a random distribution of enzymes on the lattice, waiting for macroscopic systems to evolve, similar to the *primordial soup* project. On the other hand, one might think of elaborate organisms capable of digestion, DNA transcription, etc and start with these.
A fundamental enzyme `digest` already exists. Applied to an object *o*, it rearranges the links in a way such that each object of a (sub-) system is connected to that object *o*. This ensures direct access to each object, and can immediately be used when it is needed.

The solutions to technical problems are often found in nature. Integrating evolution strategies into the system theoretical framework, the enzymatic game of life might give solutions to problems of many disciplines, functioning as a universal problem solver.

Appendix A

Complete List of Classes

The number of child classes is given in parenthesis.

```
class AdjointInfo

class SyclClass (3)
  class SyclObjectClass
  class SyclReferenceClass
  class SyclValueClass (1)
    class SyclAlgebraicClass (1)
      class SyclPredicateClass

class LocalCostFunctional

class SyclEnzymeComposite

class SyclGInterp

class SyclPrefs

class SyclWindow

class SyclCanvas
```

```
class OPT

class VPT

class SyclId (2)
  class SyclRadical
  class SyclValence

class SyclInitialize

class Kaczmarz

class SyclDoublyLinkedList (4)
  class SyclLibraryOfEnzymes
  class PredicateLibrary
  class SyclLibraryOfEnzymes
  class PredicateLibrary

class SyclTImplicitList

class SyclMapCore (4)
  class SyclBasicFunction
  class SyclMultiplyByReal
  class SyclRealFunction (1)
    class SyclBasicRealFunction (1)
      class SyclPower
  class SyclSmoothStepFunction

class SyclMembrane (1)
  class SyclStepMembrane

class SyclNode (1)
  class SyclEnzymaticNode

class SyclNode (1)
  class SyclEnzymaticNode

class SyclObject (3)
  class SyclEnzyme (54)
    class SyclAdaptFork
    class Dirac
    class SyclPropagatingDirac
    class SyclAffineBeinEnzyme
    class SyclInverseAffineBeinEnzyme
    class SyclEnzymeChain
    class SyclCleanAgendaEnzyme
    class SyclConnectBranchedPathEnzyme
    class SyclCopyObjectEnzyme
    class SyclCopyObjectsValueToRadicalEnzyme
    class SyclDeclareRootEnzyme
    class SyclUndeclareRootEnzyme
    class SyclDecodeEnzyme
    class SyclDeleteRadicalEnzyme
    class SyclDotEnzyme
    class SyclHelloEnzyme
    class SyclMakeFundamentalAdjointEnzyme
    class SyclInternalLoopEnzyme
    class SyclMainEnzyme
    class SyclMakePathEnzyme
    class SyclAssembleSystemEnzyme (1)
      class SyclLinkSystemEnzyme
    class SyclRecoverSystemEnzyme
    class SyclMakeSystemEnzyme
    class SyclValenceMultEnzyme
    class SyclValenceMultLeftEnzyme
    class SyclAdjointMultRightEnzyme
    class SyclOutPathSuccEnzyme
    class SyclOutPathLinkEnzyme
    class SyclOutPathPrecEnzyme
    class SyclOutputEnzyme
    class SyclRemovePresentationEnzyme
    class SyclCreatePresentationEnzyme
    class SyclSystemPresentationEnzyme
    class SyclPresentRadicalEnzyme
    class SyclPresentSourceEnzyme
    class SyclProgram
    class SyclProlongPathEnzyme
    class SyclPushOutsideMembraneEnzyme
    class SyclPushInsideMembraneEnzyme
    class SyclForAllValences
```

```
    class SyclForAllTriangles
    class SyclUntil
    class SyclRemoveValenceIfEnzyme
    class SyclRemoveValenceOpenTriangleEnzyme
    class SyclRemoveThisValenceEnzyme
    class SyclRenderFundamentalEnzyme
    class SyclRenderVirtualEnzyme
    class SyclReportToAgendaEnzyme (9)
      class SyclAdaptForkEnzyme (1)
        class SyclPrepareAssemblyForkEnzyme
      class SyclClimbTreeForkEnzyme
      class SyclActOnLeavesEnzyme
      class SyclBackwardForkEnzyme
      class SyclBacktrackEnzyme
      class SyclBackwardForkEnzyme
      class SyclClimbTreeForkEnzyme
      class SyclDeleteReportEnzyme
      class SyclReportSourceToAgendaEnzyme
    class SyclShowEnzyme
    class SyclSourceActsEnzyme
    class SyclSourcesRValueAddedEnzyme
    class SyclSpecifyPrincipalPortEnzyme
    class SyclVValEnzyme
    class SyclVAdjEnzyme
  class SyclSystem
  class SyclSystem

class Id (1)
  class pElement (2)
    class pAtom (8)
      class pInt
      class pDouble
      class pString
      class pError
      class pFunction
      class pObject
      class pValence
      class pEnzyme
    class pList
class ElPtr

class SyclPathLink (4)
  class SyclPathBranching
  class SyclAntiPathLink
  class SyclAntiPathLink
  class SyclAntiPathLink

class SyclPresentationFactory (2)
  class SyclQtPresentationFactory
  class SyclTextPresentationFactory

class SyclPresentation (2)
  class SyclGraphics
  class SyclText

class SyclRadicalPresentation (2)
  class SyclRadicalQtPresentation
  class SyclRadicalTextPresentation

class SyclValencePresentation (2)
  class SyclValenceQtPresentation
  class SyclValenceTextPresentation

class SyclValuePresentation (2)
  class SyclValueQtPresentation (4)
    class SyclBMatrixQtPresentation
    class SyclIMatrixQtPresentation
    class SyclDMatrixQtPresentation
    class SyclStringQtPresentation
  class SyclValueTextPresentation

class SyclProRadical

class SyclRadList

class SyclValList

class Node
```

```
class SyclEnzymeRep

class MapCoreRep

class StringRep

class SyclPropertyPlusLink

class MyClass

class SyclValue (3)
  class SyclPredicateRep
  class SyclAlgebraicValue (6)
    class SyclEnzymeValue
    class SyclListValue
    class SyclMap (1)
      class SyclRealMap
    class SyclMatrix (4)
      class SyclAffine
      class SyclAffineTransformation (1)
        class SyclAffineFrame
      class SyclDouble (1)
        class SyclMaxPlus
      class SyclInt
    class SyclPredicate (2)
      class SyclKey (1)
        class SyclAntiKey
      class SyclTrueForAllValences
    class SyclStringValue
  class SyclCoVector (2)
    class SyclAffineVector (1)
      class SyclPosition
    class SyclCoVector3d
```

Appendix B

SyCL Coded in XML

The SyCL Document Type Definition

The following dtd defines the tags *system, object, valence* and *value.* For a deeper insight in XML, see [12]

```
<!ELEMENT system (object|valence)+>

<!ELEMENT object (value)*>
<!ATTLIST object
        ID ID #REQUIRED>

<!ELEMENT valence (value)*>
<!ATTLIST valence
        ID      ID #REQUIRED
        SOURCE  ID #REQUIRED
        TARGET  ID #REQUIRED
        ADJOINT ID #REQUIRED>

<!ELEMENT value EMPTY>
<!ATTLIST value
        TYPE    CDATA #REQUIRED
        CONTENT CDATA #REQUIRED>
```

A Sample System Coded in XML

```
<?XML version="1.0"?>
<!DOCTYPE SyCL SYSTEM "system.dtd">
<system>

<object ID="2" POS="{-139,31,0}" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="0 " />
</object>

<object ID="3" POS="{18,45,0}" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="0 " />
</object>

<object ID="6" POS="{33,-51,0}" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="0 " />
</object>

<valence ID="4" SOURCE="3" TARGET="2" ADJOINT="5" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</valence>

<valence ID="5" SOURCE="2" TARGET="3" ADJOINT="4" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</valence>

<valence ID="7" SOURCE="6" TARGET="3" ADJOINT="8" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</valence>

<valence ID="8" SOURCE="3" TARGET="6" ADJOINT="7" >
  <value TYPE=" MAT" DIMM="1" DIMN="1" CONTENT="1 " />
</valence>

</system>
```

Appendix C

The Common Lisp Object System

Lisp as the favoured programming language for artificial intelligence is a very powerful tool. Its main advantages over other languages are a clear and elegant structure when dealing with recursion as well as list processing. For two reasons, Lisp was not appropriate for implementing the theory of complex systems. First of all, lists are not systems (though there exists an isomorphism between a list and a linear system)! Second, at the beginning of the implementation, Lisp was not object oriented.
The first reason still holds. From this point of view, the SyCL project can be regarded as a System Processor, which will be most powerful, when combined with the benefits of Lisp. This thought actually brought up the idea to use the approved Lisp syntax for the SyCL interpreter language, in order to be extended in the future.
As far as the second reason is concerned, the ANSI Common Lisp has been extended by object oriented features, CLOS (the Common Lisp Object System). Though it is still not suitable for an implementation of systems theory, it makes Lisp really interesting. Aside of the well known representatives of object oriented programming languages, like *C++* and *Java*, Lisp is still characterized by its simple and consequent syntax, which yields flexibility, for which it is worth being mentioned here.

The Lisp Syntax

In Lisp, everything is a *list* or an *atom*, where the list elements are either lists or atoms. As a consequence, functions are called via lists, where the first element is the name of a function, the others serve as parameters. A simple square function can be defined as

```
(defun square (n)
          (times n n))
```

`defun` is the command that defines a function, `square` is the functions name, and `(n)` is the lambda list (containing parameter names to be substituted by real parameters during the function call), followed by the functions body. The result of the function defined this way is that of the functions body. The new function can be called by

```
(square 7)
```

which of course results in 49. Lisp treats its variables by making use of the lambda calculus [25]. Alternatively one can write

```
((lambda (n) (* n n)) 7)
```

Class Definitions

Since the class definition is nothing more than the evaluation of a list, classes can be defined at runtime! This is a great advantage of CLOS compared to most other object oriented languages.

A basic class can be defined in the following way:

```
(defclass class-name ({superclass-name}*)
                     ({slot-specifier}*))
```

An expression *this** means: Zero or more occurrences of *this*. A superclass name has to be specified to derive from a base class. The slot specifiers are of the form `(slot-name slot-option*)`. The most useful slot options are:

```
:ACCESSOR function-name
:INITFORM expression
:INITARG symbol
```

The accessor defines the name, a data member can be accessed by. The initform sets an initial value when instances of that class are created. The initarg option allows to specify the initial value during instantiation.
Methods are defined via:

```
(defmethod method-name specialized-lambda-list body)
```

The specialized lambda list carries the parameter names, optionally extended by a class specification. As an example, a simple system class library may start with the definition of objects and valences:

```
(defclass object () (
      (val :accessor vlist
           :initform ())
      )
)

(defclass valence () (
      (src :accessor source
           :initform ()
           :initarg :source)
      (adj :accessor adjoint
           :initform ()
           :initarg :adjoint)
      )
)

(defmethod link ((o1 object) (o2 object))
      (setq v1 (make-instance 'valence :source o1))
      (setq v2 (make-instance 'valence :source o2 :adjoint v1))
      (push v1 (vlist o2))
      (push v2 (vlist o1))
)

(defmethod dump ((o object))
      (print o)
      (mapc 'print (vlist o))
      NIL
)
```

The objects' attributes are a list containing valences. The valences' attributes are the source object and an adjoint valence. Objects are created by the use of the `make-instance` function, while valences should be created by the use of the `link` method:

```
(setq a (make-instance 'object))
```

```
(setq b (make-instance 'object))
(setq c (make-instance 'object))

(link a b)
(link a c)
(link b c)
```

These commands create a system of three objects, linked together. The dump method implements a way to get a test output of the created elements. For example, (dump a) might return the addresses of the object *a* and the two valences connected to it:

```
#<OBJECT #x2029EA6D>
#<VALENCE #x202A8D5D>
#<VALENCE #x202A0ED1>
NIL
```

Appendix D

The SyCL Scripting Language

The syntax of the SyCL Scripting Language is very similar to the LISP syntax. To summarize the most important rules:

- Each expression can be either elementary (an *atom*), or a list.
- A list is a number of expressions, enclosed by parenthesis.
- Evaluating a list, the first element is treated as a function, while the other elements are treated as parameters and are usually evaluated first.
- To avoid evaluation, an expression may be quoted, i.e. it begins with a single quote.

Implemented Functions

The commands, this interpreter can execute are listed below. Enzymes can be applied as well, taking an object or a list of an object and a valence as a parameter.

- **(adjoint v)** returns the adjoint valence to valence v

```
: (seto a 3)
: (seto b 4)
: (setv v 1 1 a b)
: (setq vadj (adjoint v))
1
```

- **(append expr li)** appends the expression *expr* to the list *li* and returns this list. The original list remains unchanged!

```
: (setq li '(this is a list))
'(this is a list)
: (setq li1 (append 'again li))
'(this is a list again)
: li
'(this is a list)
: li1
'(this is a list again)
```

- **(car li)** returns the first element of the list *li*

```
: (setq li '(this is a list))
'(this is a list)
: (car li)
this
```

- **(cdr li)** returns the list *li* without its first element

```
: (setq li '(this is a list))
'(this is a list)
: (car li)
(is a list)
```

- **(connectto a x y z)** searches the system for an object at the position (x, y, z) and assignes the variable a to it.
- **(delete a)** removes element a from the variable list and deletes it (compare *remove*).
- **(dump)** dumps the elements from the variable list to std.out.
- **(edit a b)** changes the value of an object or valence a to the SyCL value b.

- **(enzyme expr)** defines a new enzyme from the expression *expr*. *expr* has to be of the form *xxxx=enz1enz2enz3....* and must be quoted in order not to be executed as a command! The newly defined enzyme is automatically inserted into the enzymelist as well as the enzyme menu from the GUI.

```
: (enzyme 'move=_AMR_RMV{?nAD}_PRS_RFU)
()
: (enzymes)
(_DOT _OUT _DEC _MVV _CPO _RMV _RFU _MAD _VML _AMR _PRS
_Rag _Sag _DEL _POM _PIM aDEL aFRK ... sFRK move)
```

- **(enzymes)** returns the list of enzymes

```
: (enzymes)
(_DOT _OUT _DEC _MVV _CPO _RMV _RFU _MAD _VML _AMR _PRS
_Rag _Sag _DEL _POM _PIM aDEL aFRK ... sFRK)
```

(The exact output of course depends on the set of enzymes generated by SyCL and by the user.)

- **(eval expr)** evaluates the expression *expr*

```
: (setq li '(plus 3 4))
'(plus 3 4)
: (eval li)
7
```

- **(equal a b)** or **(eq a b)** returns 1 if *a* and *b* are equal, an empty list otherwise

```
: (setq a 3)
3
: (equal a 3)
1
: (if (equal a 3) ((setq b 4)))
()
: b
4
```

- **(for (n i0 i1) (expr1 expr2 expr 3 ...))** evaluates the expressions *expr1, expr2* and *expr3* ... with the variable *n* taking integer values from *i0* to *i1*

```
: (for (n 1 3) ((seto a n)))
```

- **(funlist)** returns the list of implemented functions

```
: (funlist)
(plus + minus - times * eval setq set oblist funlist car
cdr nth replacenth remove delete load seto setv setl
enzymes dump for list newo equal eq notequal neq lessthan
lth greaterthan gth if edit makeenz enzyme adjoint append)
```

- **(greaterthan a b)** or **(gth a b)** returns 1 if *a* is greater than *b*, an empty list otherwise

```
: (greaterthan 4 3)
1
```

- **(if expr (expr1 expr2 expr3 ...))** evaluates the expressions *expr1, expr2* and *expr3* ... if *expr* is true

```
: (if (gth 4 3) ((seto a 3) (seto b 4) (setv v 1 1 a b)))
```

- **(length li)** returns the number of elements of the list *list*.
- **(lessthan a b)** or **(lth a b)** returns 1 if *a* is less than *b*, an empty list otherwise

```
: (lessthan 4 3)
()
```

- **(list a b ...)** returns a list with the parameters *a, b, ...* al elements

```
: (setq li (list 3 4 5 6))
(3 4 5 6)
: li
(3 4 5 6)
```

- **(load filename)** loads and evaluates commands in a file *filename*. This can be done more comfortably with a file-select box via the GUI *open file* menu entry.

```
: (load 'test.syp)
```

- **(minus a b) (- a b)** returns the difference of *a* and it b

```
: (minus 3 5)
-2
```

- **(newo a)** creates a new SyCL object of SyCL value *a* and returns an object element

```
: (newo 3)
3
```

- **(notequal a b) or (neq a b)** returns an empty list if *a* and *b* are equal, 1 otherwise

```
: (notequal 3 4)
1
```

- **(nth n li)** returns the *n*th element of a list *li*. Counting starts at zero!

```
: (setq li '(this list has five elements))
'(this list has five elements)
: (nth 3 li)
five
```

- **(nthval n a)** returns the *n*th valence of the valences directed toward the object *a* (counting from 0).
- **(oblist)** returns the list of variables
- **(plus a b) (+ a b)** returns the sum of *a* and *b*

```
: (plus 3 4)
7
```

- **(rand n)** returns an integer random number between zero (inclusive) and *n* (exclusive).

- **(remove a)** removes element *a* from variable list (compare *delete*).
- **(replacenth n li expr)** replaces the *n*th element of a list *li* by *expr* and returns this list. The original list *li* remains unchanged!

```
: (setq li '(1 2 3 4))
'(1 2 3 4)
: (setq li1 (replacenth 2 li 'three))
'(1 2 three 4)
: li
'(1 2 3 4)
: li1
'(1 2 three 4)
```

- **(run '(e v))** lets the enzyme *e* act on the source of valence *v*.
- **(set a b)** same as **setq**, but evaluates *a* first

```
: (setq a 'b)
b
: (set a 8)
8
: a
b
: b
8
```

- **(setl l v a b)** *a* and *b* have to be object elements. An unidirectional valence is created between *a* and *b* with a SyCL values *v*. The valence element is assigned to the variable *l*

```
: (setq a 3)
3
: (setq b 5 100)
5
: (setv v 1 1 a b)
1
```

- **(seto a ['int/'real/'bool/'enz] ['vector m/'covector n/'matrix m n] b... [x [y [z]]])** creates a SyCL object with a SyCL value *b* (evaluated) and assigns the object element to the variable *a*. Creating a vector of dimension m, a covector of dimension n, or a matrix of dimension *mxn* requires *mxn* elements *b*! *x, y, z* can be appended to specify coordinates.

```
: (seto a (plus 3 4))
7
: (seto b 'int 'vector 3  4 5 6)
: b
4 5 6
```

- **(setq a b)** creates an expression, sets it to *b* (evaluated) and assigns it to the variable *a*

```
: (setq a (times 3 4))
12
: (oblist)
(a)
```

- **(setv v v0 v1 a b)** *a* and *b* have to be object elements. A bidirectional valence is created between *a* and *b* with SyCL values *v0* and *v1* for the valence and its adjoint. The valence element is assigned to the variable *v*

```
: (setq a 3)
3
: (setq b 5 100)
5
: (setv v 1 1 a b)
1
```

- **(times a b)** or **(* a b)** returns the product of *a* and *b*

```
: (times 3 4)
12
```

- **(valences a)** returns the number of valences directed toward an object *a*.

Examples

- Example 1:
To create two objects (with values 3 and 4, at different positions) and a valence (with values 1 and 1), type:

```
(seto a 3  100)
(seto b 4  0 100)
(setv v 1 1 a b)
```

- Example 2:
To create a ring of seven objects linked by valences, type:

```
(seto a 0)
(setq c a)
(for (x 1 6) (
  (seto b x)
  (setv v 1 1 c b)
  (setq c b)
  (if (eq x 6) (
    (setv v 1 1 c a)
    )
  )
  )
)
```

To clean up the variable list, type:

```
(remove b)
(remove c)
(remove v)
```

- Example 3:
A simple 4x4 lattice is created by the following program:

```
(setq li '())
(for (y 0 3)
  (
    (for (x 0 3)
      (
        (seto a x (* x 40) (* y 40))
        (if (neq x 0)
          (
            (setv v 1 1 a ax)
            )
          )
        (setq ax a)
        (setq li (append ax li))
        (if (neq y 0)
          (
            (setq ay (car li))
            (setq li (cdr li))
            (setv v 1 1 a ay)
            )
          )
        )
      )
    )
  )
```

- Example 4:
  To apply an enzyme to an object:

```
(seto a 8)
(_CPO a)
```

- Example 5:
  The lattice used for the cellular automaton (see chapter 6) is set up by the following script:

```
(setq li '())
(for (y 0 5)
  (
    (for (x 0 5)
      (
        (seto a 0 (* x 60) (* y 40))
        (seto b 0 (* x 60) (* y 40) 80)
        (setl 1 1 b a)
        (if (neq x 0)
          (
            (setv v 1 1 a ax)
```

```
          ))
        (setq ax a)
        (setq li (append ax li))
        (if (neq y 0)
          (
            (setq ay (car li))
            (setq li (cdr li))
            (setv v 1 1 a ay)
          )
        )
      )
    )
  )
)
(enzyme 'dist=aFRKfALL(_AMR{?nAD}){?iLK&&?iAD})
(enzyme 'ccam=aFRK_CAM)
```

- Example 6:
  In the following example, a system of three objects is created. The object *a* is connected to both of the other objects. (see figure D.1).

```
(seto a 1)
(seto b 2 100)
(seto c 3 0 100)

(setv v 1 1 a b)

(setv m 1 1 a c)
(setq n (adjoint m))

(enzyme 'move=_AMR_RMV{?nAD}_PRS_RFU)
```

The enzyme move applied to the object *a* and a valence (direction) *m* forces all valences except for *m* pointing toward *a* to move to the object *c* which is the source of *m*. To see this happen, type:

```
(move '(a m))
```

This can of course be done in the opposite direction as well. Type:

```
(move '(c n))
```

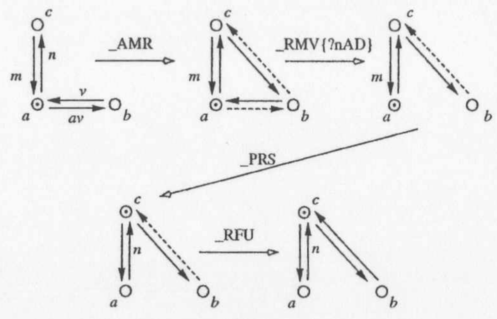


Figure D.1: Visualization of Example 6

Appendix E

THE ENZYMATIC GAME OF LIFE

The script to set up the universe is listed below:

```
(setq elist
      '(sFRK _CPO _VML _AMR _MVV _MAV _PRS _RFU _MAD _CPO _CPO _RMV))
(setq enum (length elist))

(setq li '())

(for (e 0 200) (
  (setq x (- (rand 500) 250))
  (setq y (- (rand 400) 200))
  (seto a 'enz (nth (rand enum) elist) x y)
  (setq ax a)
  (setq li (append ax li))
))

(setq lilen (length li))

(for (v 0 200) (
  (setq o1 (nth (rand lilen) li))
  (setq o2 (nth (rand lilen) li))
  (setv w 1 1 o1 o2)
))
```

An example for the dynamics might look like this:

```
(for (t 0 2000) (
  (setq obj (nth (rand lilen) li))
  (setq val (nthval (rand (valences obj)) obj))
  (run '(obj val))
))
```

Appendix F

THE SYCL USER INTERFACE

To get an impression of the graphical user interface, a screenshot is shown in figure F.1. Due to constant development, the intructions how to use this software can be found on the SyCL webpage [18].

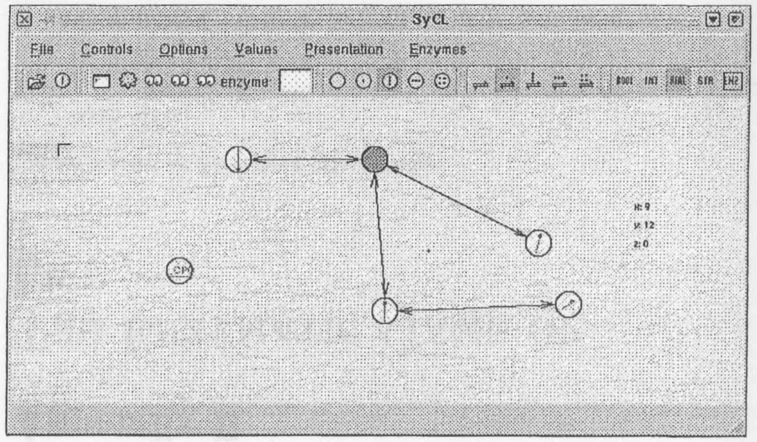


Figure F.1: The SyCL main window.

Appendix G

A Philosophical Remark

It is interesting to see how the ancient greek philosophers have influenced todays way to view the world. Not only Demokrit participated in physics, establishing the particle view of the world by introducing atoms. Also Plato and Aristotle, though many of their ideas have been rather contradictary, had a subtle feeling for a natural way to handle complexity.

Platos concept of ideas resembles the modularity aspects of the object oriented paradigm by stating that an object is not defined by its concrete properties rather than by an abstract idea: It is its functionality that lets an object be recognized as such. Thus ideas may exist, without the necessity of any existing instances.
This also interlaces the concept of encapsulation: You can use a tool, when you know the idea of that tool, even if you dont know how it works internally.

The way Aristotle viewed the ideas is even closer to the system theoretic point of view, pointing out that the human mind creates ideas of objects by experience. Stated system theoretically: A composed object is identified as an object, when it behaves like an object, or just: Structure is in the eye of the beholder.
Applying this concept of composite objects (not only to concrete objects but) to their abstractions (Platos ideas), results in an abstract describtion of the inheritance aspect of the object oriented paradigm. Also this was done by Aristotle. He said that the human mind identifies complex (com-

posite) objects by combination of ideas.

This shows once more that many concepts of modern theories have a very long tradition. Today we have the technology to apply them to computer simulations.

Bibliography

[1] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J. Watson. Molecular Biology of The Cell. Garland Publishing, Inc. 1989.

[2] G. Booch. Object-Oriented Analysis and Design. Addison-Wesley, 1994.

[3] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[4] E. Gamma, R. Helm, R. Jahnson, J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[5] M. de Groot, The Primordial Soup, http://alife.santafe.edu/alife/software/psoup.html, Santa Fe Institute 1995.

[6] S. Gößling. Approximations to the Renormalization Group Flow of the Sine-Gordon Model - A Comparative Study. Masterthesis, Hamburg 1997.

[7] D. Harel and E. Gery. Executable Object Modeling with Statecharts. IEEE Computer Society. Computer V.30, #7, July 1997.

[8] K. H. Hoffmann, M. Schreiber. Computational Physics, Selected Methods, Simple Exercises, Serious Applications. Springer, 1996.

[9] P. James-Roxby. Advanced Software Engineering. http://www.eee.bham.ac.uk/dsvp_gr/roxby/ee4a3/Lecture2/sld001.htm

[10] A. Kiciman. Grammed: A Generic Grammar Editor for Graph-based Visual Languages. Masterthesis, Geneva 1997

[11] S. Levy. Artificial Life. Pantheon Books, 1992.

[12] R. Light. Presenting XML. Sams.net Publishing, 1997.

[13] G. Mack. Gauge Theory of Things Alive: Universal Dynamics as a Tool in Parallel Computing. Progress of Theoretical Physics, Supp.#122, 1996.

[14] G. Mack. Universal Dynamics of Complex Adaptive Systems: Gauge Theory of Things Alive (DESY-94-075) (1994).

[15] G. Mack. Gauge theory of things alive, *Nucl. Phys.* B42:923 (1995)

[16] G. Mack. Pushing Einsteins Principles to the Extreme, in: *Quantum Fields and Quantum Space Time*, edited by G. tHooft et al., New York (1997). Plenum Press. NATO-ASI series B: Physics vol. 364 (preprint gr-qc/9704034).

[17] G. Mack. Multigrid Methods in Quantum Field Theory. Cargese lectures July 1987.

[18] G. Mack. Theory of Complex Systems Webpage, http://lienhard.desy.de.

[19] M. Meier-Schellersheim and G. Mack. SIMMUNE: A tool for simulating and analyzing immune system behaviour, preprint DESY-99-034, University of Hamburg (1999), submitted to J. Theor. Biol.

[20] K. J. M. Moriarty, S. Sanielevici, K. Sun and T. Trappenberg. Object Oriented Programming Applied to Lattice Gauge Theory. Computers in Physics, Vol. 7, No. 5 (1993) 560.

[21] NeXT. Objective-C Manual, OPENSTEP documentation package, NeXT Software Inc., (1996).

[22] B. Oestereich. Objektorientierte Softwareentwicklung mit der Unified Modeling Language. Oldenbourg, Wien (1997).

[23] A. D. Sokal. How to Beat Critical Slowing Down, 1990 Update, Nucl. Phys. B (Proc. Suppl.) 20 (1991).

[24] M. Speh. Object-Oriented Literate Design of a Multigrid Toolkit for Lattice Gauge Theory Simulations, Phd-thesis, University of Hamburg (1994).

[25] D. Rathje. Die Theorie der Berechenbarkeit, Einführung in Maschinen, rekursive Funktionen und den λ-Kalkül, unpublished talk, 1999.

[26] T. Toffoli, N. Margolus. Invertible Cellular Automata: A Review. Physica D 45, 229-253, 1990.

[27] T. Tomkos. Formal Specification and Modeling of Complex Systems - Towards a Physics of Information via Networks, Phd-thesis, University of Hamburg (1999).

[28] G. Vogel/H. Angermann. dtv-Atlas Biologie, BI. Deutscher Taschenbuch Verlag GmbH & Co. KG, Muenchen 1984.

[29] K.G. Wilson and J. Kogut. The Renormalization Group and the ϵ Expansion, Physical Reports 12, No.2 (1974).

[30] K. G. Wilson. Ab Initio Quantum Chemistry: A Source of Ideas for Lattice Gauge Theorists, Nucl. Phys. B Proc. Suppl. 17 (1990).

[31] J. Würthner. Aspekte des Renormierungsgruppenflusses des Sine-Gordon Modells. Masterthesis, Hamburg 1997.

ACKNOWLEDGMENT

Without the influence of certain people this work would have turned out totally different. Their support, technical as well as social, has been of unmeasurable value for which I want to express my very special thanks.

Prof. Mack has given the very interesting subject and taken care of the progress of my work. He always took his time to discuss on my subject and we have been working close together.

The whole Mack group, Michael Bartels, Martin Meier-Schellersheim, Ute Kerres, Dirk Rathje, Claudia Lehmann, Mathias de Riese and Thorsten Prüstel supported me with numerous discussions on systems theory as well as other interesting topics during our inofficial seminar. My parents supported me in many ways. My brother Jens made me see what its like to have a brother *(HMDS)!*

The Linux community confirmed me in my belief that an operating system *can* be stable. I enjoy developing for a system with heart.

With the internal choir of building 2a, *Spontane Harmoniebrecher*, we proved the strong correlation between physics and music on tuesdays (cp. the *Tuesday Effect*). Also *Chor Hagenbeck* deserves my very polyphonical thanks for showing me how singing can make a poor man happy.

Herr Graesslin has brought up valuable questions from philosophy. I enjoyed the discussions in the foyer of our seminar room. Oliver Gumtau impresses me not only by thinking the unthinkable and questioning the unqestionable. Fabian Wenzel has been my companion in studying complex object oriented software architechtures. I enjoyed working on our famous sequencer program. Tim Ulmer proved that the length of a telephone call is not at all correlated to the distance he is calling from. I also enjoyed the programing sessions with Tilo Ermlich late at night. Martina Hausen deserves my deepest apologies. I have not written a single letter, postcard, or email in years - and she still forgives me. Matthias Töwe has moved to a distant country and is still a very close friend of mine. This will never change. Thomas Tomkos has been a close companion through many years. Exchanging thoughts on sciences, philosophy, religion, music, and of course humor has become a very large part of my life. Our accidental discussions have been as interesting as enjoyable.

My dear Annette Kirchner deserves all my love. Thank you so much for being the honest and warm hearted person you are.

Finally, I acknowledge the Deutsche Forschungsgemeinschaft (DFG) for the financial support.