A Simple Artificial Chemistry Experiment Set

DIPLOMA 2006

Berne University of Applied Sciences
School of Engineering and Information Technology
Departement Information Technology

Authors
Anthony Aguillon, Daniel Noelpp

Supervisors
Dr. Peter Schwab, Dr. Thomas Hinze

Expert
Prof. Dr. Federico Flueckiger

Abstract

A simple experiment tool in *artificial chemistry* is proposed, namely 'saces'. An artificial chemistry is a man-made system that is similar to a real chemical system. Usually it consists of a population of molecules, a set of reaction rules, and an algorithm which executes the reactions. It doesn't try to find most realistic models of chemistry. Nature science laws are abstracted and simplified.

The tool 'saces' does ideal gases in a rectangular reaction vessel. Molecules are modeled as hard spheres. Collisions are elastic if no reactions apply. The particles are animated in a three-dimensional visual display. 'saces' is intended both as an application in artificial chemistry and as a simple educational tool.

Contents

1 Introduction **5**

1.1 Subject of the thesis . . . 5
1.2 Task description . . . 5
1.3 Aims and intentions . . . 5
1.4 Fields of the thesis . . . 5
1.5 Software and Hardware . . . 6
1.6 Acknowledgments . . . 6

2 What is Artificial Chemistry? **7**

2.1 Rationale . . . 7
2.2 An Example: The SAT Problem . . . 8
2.3 SAT and Artificial Chemistry . . . 10
2.4 The general form of an Artificial Chemistry . . . 10
2.4.1 The molecules . . . 11
2.4.2 The reactions . . . 11
2.4.3 The algorithm . . . 12
2.5 Applications of Artificial Chemistry . . . 12
2.6 Summary . . . 13

3 Proposal of ‘saces’ as a tool in Artificial Chemistry **14**

3.1 Fact Sheet . . . 14
3.2 The position of ‘saces’ in Artificial Chemistry . . . 15
3.2.1 Abstractions and Simplifications . . . 15

3.3 General form of the Artificial Chemistry of 'saces' 17
3.3.1 The molecules 17
3.3.2 The reactions 17
3.3.3 The algorithm 18
3.4 A short overview about the 'saces' tool 18
3.4.1 The three-dimensional view of the reaction vessel 18
3.4.2 The simulation loop 18
3.4.3 The data viewer 19
3.4.4 The experiment settings dialog 20
3.4.5 Saving simulation data 20
3.5 Getting started with 'saces' 21
3.5.1 Getting 'saces' 21
3.5.2 Java Runtime Environment version 5 21
3.5.3 Optional: Getting JOGL 21
3.5.4 Start 'saces' 22
3.5.5 Stop simulation and load experiment 22
3.5.6 Tweak the experiment 22
3.5.7 Start the new experiment 23
3.5.8 See the binary log 23
3.5.9 The snapshot 23

4 The User Manual **24**
4.1 Overview 24
4.1.1 The main view 24
4.1.2 The settings window 24
4.2 Navigation of the three-dimensional view 25
4.3 Plug-and-play architecture 25
4.4 Compiling 'saces' 25

5 Some Experiments with 'saces' 28

5.1 The sample experiment 28

5.2 A chemical explosion 29

5.3 A Lotka-Volterra system 29

5.4 Brownian movement 29

5.5 A finite automaton 29

6 The Internals of the tool 'saces' 30

6.1 Internal structure 30

6.2 Data architecture 31

6.2.1 Experiment setup 31

6.2.2 Simulation State 33

6.3 The process architecture 36

6.3.1 The mediator 36

6.3.2 The simulation process 37

6.3.3 The binary logger 46

6.4 Handling Time 46

6.5 An optimization: Space Partitioning 46

6.6 How to implement the plug-and-play classes 46

6.7 OpenGL 47

6.8 Todo: Other sections? 47

Chapter 1

Introduction

(todo: This chapter is a copy of the diploma specification according to the school. We need to talk with the supervisors about it later.)

1.1 Subject of the thesis

(todo: Use the blurb in the diploma specification.)

1.2 Task description

(todo: Use the blurb in the diploma specification.)

1.3 Aims and intentions

(todo: Use the blurb in the diploma specification.)

1.4 Fields of the thesis

- Artificial Chemistry
- Chemistry
- Physics

- Java Software Engineering
- OpenGL
- Software Profiling and Optimization

1.5 Software and Hardware

Software Java 5 and JOGL.

Hardware Platform neutral, but rather ressource-intensive. ‘saces’ needs a video card with OpenGL hardware acceleration and enough RAM (at least 512 MB).

1.6 Acknowledgments

It is a pleasure to acknowledge the assistance of several people who helped, supervised or gave feedback or support in any form for our diploma. We especially thank Thomas Hinze from the TU Dresden, Peter Schwab from the Berne University of Applied Sciences and Federico Flueckiger from the University of Applied Sciences of Southern Switzerland for their most valuable input to the project. We thank . . . , . . . , Matthias Noelpp that they read the first concoctions of the diploma with a critical eye and found many mistakes. Then finally we thank our friends and relatives for their unrelenting support.

Chapter 2

What is Artificial Chemistry?

2.1 Rationale

As the name already suggests, an *artificial chemistry* is a constructed, artificial world. Nature science laws of physics and chemistry are prototypes of the rules in an artificial chemistry, that means, there is no attempt to emulate nature in utmost detail.

The famous *Conway Game of Life* [Sil05] follows very simple rules and yet exhibits very complex patterns and forms of quasi-life and is even Turing Complete[1]. The rules of *Game of Life* are very highly abstracted and removed from nature science laws and rules. This is a very extreme example. Artificial chemistry on the other hand models a reaction vessel with reactions, but does not try to model the real world the most exact possible, as does *Computational chemistry*, for example.

In [DZB01], page 227, a broad definition is given:

> *An artificial chemistry is a man-made system that is similar to a real chemical system.*

Some might ask: Why invent artificial worlds? What are the benefits of "fantasy" worlds with rules not quite the same as the "real" laws of physics?

First there is simplification. With simple models one better can focus on the questions he really wants to explore. And secondly there is the educational value.

[1] A computational or logical system is called *Turing Complete* if it has a computational power equivalent to a universal Turing Machine. In other words, the Game of Life can in principle compute everything what a computer can compute.

Students or people not very versed in a specialized and difficult topic can follow simple models and understand them better. And lastly one can compare abstracted models with real-world ones (or with experiments) and be surprised about the similiarities and differences.

In computational chemistry on the other hand some properties of molecules (reaction behaviour, molecular "shape", total energy, dipole moment, etc.) are modeled with quantum-chemical computational methods. Software has been developed which base on many methods that solve the molecular Schrödinger equation. For example a package of Fortran programs, the Molpro quantum chemistry package [WKL+03], can perform accurate *ab initio*[2] calculations for large molecules. This is complex mathematics. Large-scale numeric calculation is needed for many of these methods.

Artificial chemistry, however, is more about speculative thinking than a realistic model of the world. The aims and intentions in using artificial chemistry are more important. Someone might want to explore computing models like DNA or molecular computing. Can he solve some problems from Theoretic Computer Science in artificial chemistry? And how do the time and space complexities behave asymptotically?

In DNA computing a "Traveling Salesman" problem has been solved using real DNA [Adl94]. Adleman demonstrated with the experiment the feasibility of carrying out computations at the molecular level. It is therefore possible to solve other well-known Computer Science problems like the satisfiability of logic clauses, the so-called SAT problem, in this way.

2.2 An Example: The SAT Problem

To see the connection between artificial chemistry and real chemical reactions we take the SAT Problem as an example. First we explain the problem, then what happens if we would tackle it with a massively parallel computer using DNA- or chemical computing. Artificial chemistry can help to explore and to find computing models for such problems.

A logic clause consists of boolean variables joined together with conjunctions

[2]In quantum chemistry or quantum mechanics *ab initio* is understood as the solving of the Schrödinger equation using only the "first principles", for example using only the constants of Nature Science. For a more complete treatment of *ab initio* please see [Sce04].

and disjunctions. Some variables can be negated. An example is[3]

$$(x_1 \vee x_2 \vee x_4) \wedge\ (\neg x_3 \vee x_4 \vee \neg x_5) \wedge\ (\neg x_1 \vee \neg x_2 \vee (x_3 \wedge \neg x_5))$$

The question is: Does a solution exist which fulfils the logic clause? Every variable holds a boolean value, yes or no; true or false. Can we find an assignment of boolean values to the variables so that the logic clause becomes true?

The SAT problem is NP-complete. The only known reliable algorithm is the full enumeration of the problem space. We have to try every possible solution before we can answer for sure: "No, this logic clause doesn't have a satisfiable assignment". So computers of today are not very good at coping with the SAT problem. The SAT problem is important, however, because many practical problems (database queries, Artificial Intelligence and expert systems, Electronic Design Automation, etc.) depend on SAT.

The complexity (expense in either time or memory) doubles every time with an additional variable. Suppose we have a computer that can solve a SAT problem with 25 variables in about one second. If we give it a SAT problem with twice as many variables, namely 50 variables, it will need about a year to find an answer! (We assume about 30 nanoseconds for a step and about 33 million steps for 25 variables and these 33 million steps squared ($1.12 \cdot 10^{15}$) for 50 variables.)

If we have a good computer using chemistry or molecular biology for its calculations, we can trade memory for time. Instead of trying out all possible solutions one after one, we try all possible solutions at once and extract the solutions. Such a computing model allows massively parallel algorithms. Information can be packed much, much more densely as molecules in a test tube than as electromagnetic states in RAM. And an operation can be applied to all the molecules simultaneously, for example by pouring an reagent into the test tube.

There is still an important limitation with the trading of memory for time, however.

NP-complete problems have exponential growing complexities. If we add an additional variable to a SAT problem, we double the problem space. There are twice as many possible solutions to verify. Suppose we buy a computer which works twice as fast. Then we will be able to solve SAT problems with only one more variable in the same time. A rather pathetic improvement. (todo: kill this paragraph and shorten the next one)

[3] The vee operator $\vee$ is the disjunction ('or'). The wedge operator $\wedge$ is the conjunction ('and'). And $\neg$ is the negation operator ('not').

If we had a computer based on chemistry working a million times faster, then we could add twenty more variables to the SAT problem (the dual logarithm of a million is roughly 20). This is better, but not much. If we would want to add 100 more variables, we would soon run out of available molecules. The computation would need more stuff in the test tube than the earth weighs, for example. There is just not enough "memory" to solve large SAT problems. This is the limitation with the trading of memory for time.

It is not known whether a more efficient algorithm for the SAT problem exists or not. One of the greatest unsolved problems of Mathematics and Computer Science is the "P versus NP" problem [Coo03], [Dev02]. Most experts "believe" that there are no efficient (polynomial time) algorithms for NP complete problems. But this has not been proven yet.

But a practical way to cope with the SAT problem is to use heuristics (backtrack search). In practice, the SAT problem can be solved in reasonable time for many instances [Nad02].

2.3 SAT and Artificial Chemistry

One application of artificial chemistry is to explore the possibilities and limitations of computing models without implementing the models in reality or doing exhaustive simulations.

Suppose someone designs a working computer with DNA. With simple reasoning it has been shown that even massive parallelity with many DNA strands is not "enough". Massive parallelity will just push the limit of the problem size by a few dozen units.

While this was possible with a little thinking, more complex problems are perhaps better formulated as an artificial chemistry to get answers for some questions. We suggest that analyzing computing models is one of the applications of artificial chemistry.

2.4 The general form of an Artificial Chemistry

In [SBB+00] and [DZB01] a formal definition of an *artificial chemistry* is given. A model of a reaction vessel or a domain containing objects or molecules and of reactions inside the vessel is assumed. The definition is (see [DZB01], section 2.1, page 227):

> An *artificial chemistry* can be defined by a triple (S, R, A), where S is the set of all possible molecules, R is a set of collision rules representing the interaction among the molecules, and A is an algorithm describing the reaction vessel or domain and how the rules are applied to the molecules inside the vessel.

Let us have a look at the molecules, the reactions and the algorithm.

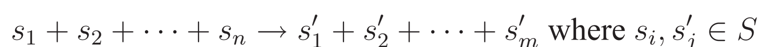
2.4.1 The molecules

The elements $s \in S$ are molecules. They can be just objects (like strings of characters A, C, G, T as highly abstracted models of DNA strands) or numbers to solve mathematical problems. To solve the SAT problem one can use boolean arrays or binary strings. A yes/no value in such a string is an assignment to a variable in the logic clause. The set S might be even infinite (as the set of natural numbers). This doesn't mean that the population of molecules inside the reaction vessel itself is infinite, only the "choice" of possible elements of S is infinite.

The molecules can have additional parameters like position or speed inside the reaction vessel. These parameters are used in the reaction rules.

2.4.2 The reactions

R, the set of reaction rules, describes the interactions between the objects in the domain. A rule $r \in R$ can be written in the same notation for chemical reactions:



$$s_1 + s_2 + \cdots + s_n \to s'_1 + s'_2 + \cdots + s'_m \text{ where } s_i, s'_j \in S$$

The elements s_i are educts of the reaction and the elements s'_j products. The educts react with each other and are removed from the population in the vessel. After the reaction the products are added to the population. The "+" sign is not a mathematical operator here, just a separator between the reagent symbols.

Rules can have conditions and, exactly as molecules, additional parameters. The most important condition of rules is that all educts must be available or have collided. An additional condition could be a reaction probability (the reaction is executed only if a random number in range $[0..1]$ exceeds reaction probability). The reaction probability is one of the additional parameters of the rule. Another rule parameter or condition could be the activation energy (the rule is activated only if the kinetic energy of the educt molecules exceeds the activation energy).

The set of rules R can be infinite like the set of molecules S. Infinitively many rules can be constructed by a meta-rule, as the rule $a + b \rightarrow (a/b) + b$, for example. a and b are natural numbers and a is divisible by b. The products are the new natural number a/b and b. Such a meta-rule is useful for an artificial chemistry to find *prime numbers* ([SBB$^+$00], page 13).

2.4.3 The algorithm

The algorithm determines which molecules react with each other, then what is done with the reaction products, and how to treat molecule and reaction parameters. A very simple artificial chemistry doesn't even have the notion of space. Molecules which "collide" which each other are selected randomly and then reaction rules are applied on the collided molecules. Others have space (especially 'saces' with its three-dimensional reaction vessel) and molecules collide when their trajectories cross each other.

Another approach is to use differential equations. Reaction rules are reformulated to contain stoichiometry factors. Such a rule can be understood as a recipe: "Add two parts of stuff A, three parts of stuff B, etc., and you get one part of stuff X and three parts of stuff Z". Instead of acting single collisions the algorithm calculates the concentration of the stuffs in the reaction vessel with differential equations (see [DZB01], page 229).

There are many different approaches to let molecules react and to handle reaction products. Sometimes an alternative definition of artificial chemistry makes more sense, namely a tuple (S, I), where S is the same as before, a set of molecules, and I a description of the interactions among the molecules. This definition avoids a separation between reaction rules and the algorithm if both are intertwined very much.

2.5 Applications of Artificial Chemistry

(todo: This section needs some more 'flesh' and restructuring. Use the material of diziba and sbbdz. Perhaps a short overview about current artificial chemistries makes sense.)

The applications of *artificial chemistry* are more important than exact modeling of chemistry using quantum mechanics. With simplified and abstracted models one can better investigate whether they are Turing Complete like the Conway Game of Life or what computing capabilities they own. The models of artificial chemistry also allow a better understanding of the chemical processes for students.

And another important application of artificial chemistry is chaos theory and the study of self-replication and life. Possible questions are:

1. How can one define Artificial Chemistries or systems within to “solve” a problem?
2. How do such systems behave dynamically?
3. Where and how do regular patterns emerge from chaotic systems?
4. When do systems collapse, for example through catastrophic positive autofeedback loops?
5. When do self-replicating patterns emerge from systems?
6. What is life?

Artificial chemistry is used as a model for a “complex system” in systemic thinking as well [SBB+00]. Systemics is an important interdisciplinary method to cope with “living” and difficult-to-manage systems. Such systems don’t behave in a linear way, have long reaction delays or intricate interconnections. Such systems are “alive”.

2.6 Summary

(todo: reformulate better!)

In a summary, artificial chemistry is very fascinating but a bit speculative. There is the danger of just “playing around” and inventing beautiful, esthetic systems of little scientific value. There is even a science fiction novel about a virtual life based on an Artificial Chemistry of only 32 chemical components, the chemical elements [Ega95].

Chapter 3

Proposal of ‘saces’ as a tool in Artificial Chemistry

As a part of the diploma a simple, small tool is proposed: ‘saces’. The name is the acronym of “Simple Artificial Chemistry Experiment System” and the code-name of the project. It does three-dimensional visual simulations of ideal gas processes. It is both useful as an educational tool and as an application of *artificial chemistry*.

3.1 Fact Sheet

The tool ‘saces’

- is a moderately realistic simulation of an ideal gas undergoing reactions.
- has a three-dimensional real-time visual animation of the chemical process.
- is useful as a simple educational tool and as an application of artificial chemistry.
- allows the specification of particles and chemical reactions.
- uses a space partioning algorithm for almost linear-time collision detection of the molecules.
- uses Monte-Carlo methods for collision response and for reactions.
- is developed in Java 5 using JOGL (see [Dav04]).

- allows replacement of steps in the simulation loop (Collision detection and response, handling reactions, etc.; see section 3.4.2 on page 18).
- runs on Windows, Linux and Mac OS X 10.4.
- needs enough RAM (512 MB minimum) and a hardware-accelerated video card with OpenGL.

Further technical details are in chapter 6, page 30.

3.2 The position of ‘saces’ in Artificial Chemistry

‘saces’ acts on a middle ground between the high abstractions of “true” artificial chemistry and the highly realistic models of computational chemistry. We wanted to simulate certain simple chemical processes and at the same time provide a nice and instructive visual display. Some corners had to be cut, or to express in other words, we had to introduce some abstractions and simplifications to reach adequate animation speeds.

Nevertheless, it is possible to do artificial chemistry scenarios as well (see chapter 5, page 28).

3.2.1 Abstractions and Simplifications

- Molecules are hard spheres. Collisions are elastic (except if reactions apply).
- The reaction vessel or test tube has flat rectangular walls. The space inside the reaction vessel has the form of an ashlar (see figure 3.1).
- Ideal gases are assumed. There are no Van der Waals forces and other intermolecular forces. For pressure measurements, however, the volume of the spheres is considered. (todo: really?!)
- No quantum chemistry.
- Collision detection is not water-tight. A few percent of all collisions are not detected. Particles overlapping to neighboring partitions are ignored in these partitions (see chapter 6, page 46).

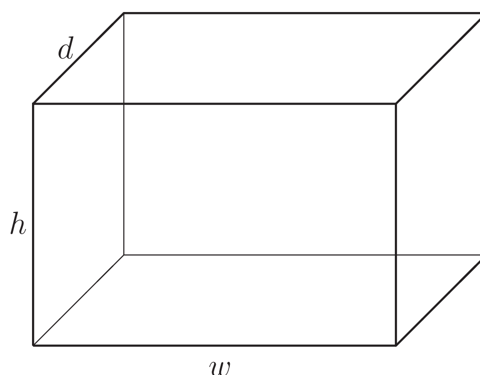


Figure 3.1: The shape of the reaction vessel in ‘saces’

- Another problem is the quasi-simultaneous collision of more than two particles. Particles after the second are ignored or handled in the next simulation loop iteration.
- Collision response determines the reflection angle randomly. Rutherford or other sophisticated reflection models aren’t implemented. (They could be programmed as an extension to ‘saces’, but perhaps at the expense of real-time speed.)
- Only these types of reaction equations are supported:

  Transform of form $E_1 + E_2 \rightarrow P_1 + P_2$ (two educts transform to two products),

  Merge of form $E_1 + E_2 \rightarrow P$ (two educts merge to one product) and

  Decay of form $E \rightarrow P_1 + P_2$ (an educt decays spontaneously to two products).

  More complicated reactions can be composed out of these simple equations.

Most of these simplifications arise from the need to do the calculations in real-time. For details how the calculations are implemented (see chapter 6, page 30).

Some limitations can be lifted if you implement your algorithms in Java code and instruct ‘saces’ to use the new classes instead (see section 4.3, page 25).

3.3 General form of the Artificial Chemistry of ‘saces’

We analyze ‘saces’ corresponding to the definition of artificial chemistry as a triple (S, R, A) (see section 2.4, page 10).

3.3.1 The molecules

The elements of the set S are particles moving around in the test tube in the form of hard spheres. They have the parameters:

- the position $\vec{p}$
- the velocity $\vec{v}$
- its “species”, namely the particle class with the parameters
  - mass m,
  - bound energy H,
  - sphere radius r, and
  - a name and a color for display

The initial number of particles and their particle classes are determined by the experiment configuration. The set S is finite, provided we ignore the molecule parameters.

3.3.2 The reactions

The rules have at most two educts and two products and have the parameters:

- reaction probability p_r and
- activation energy E_r

A reaction rule is applied only if the kinetic energy of the colliding molecules exceeds the activation energy and if the generated random number in range $[0..1]$ exceeds the probability p_r. The set R is finite as well.

3.3.3 The algorithm

The algorithm takes into account the velocities and the positions of the molecules. It does collision detection and collision response for each particle. The list of collision pairs is given to the reactions to be handled. A special case are decay reactions which occur quasi-spontaneously (with random numbers).

The visualization of the moving particles isn’t part of the definition of the artificial chemistry algorithm. ‘saces’ is not a “pure” artificial chemistry tool, but is useful as a visualization of chemical reactions, too.

3.4 A short overview about the ‘saces’ tool

The most important parts of the tool are:

- A three-dimensional view of the rectangular reaction vessel
- The simulation loop
- A data viewer to view diagrams or histograms
- A dialog to define the experiment settings and chemical reactions
- A binary logger to write data to the disk

3.4.1 The three-dimensional view of the reaction vessel

The main window displays a three-dimensional view of the test tube. Below the view a tool bar with buttons is shown. During a simulation moving spheres are shown (see figure 3.2). The user can move around or insde the test tube with navigation keys (see section 4, page 25).

3.4.2 The simulation loop

The simulation loop calculates the simulation, displays the particles and saves data to a binary file. The simulation loop consists of these steps in an endless loop till the animation is stopped:

1. Reposition the particles

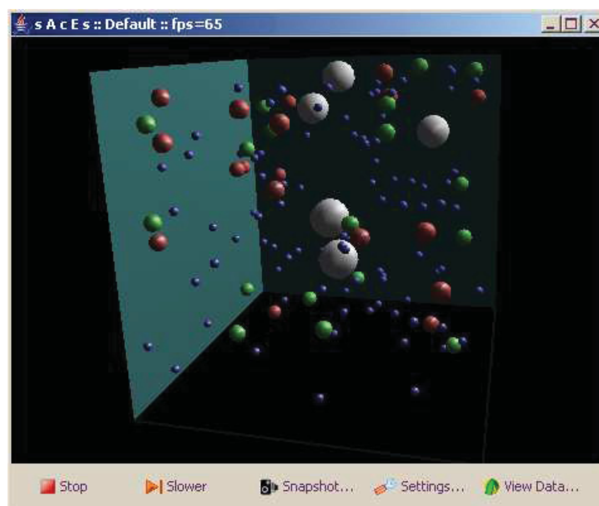


Figure 3.2: Running the sample experiment

2. Detect collisions at the test tube walls and reflect*
3. Do the measurements (pressure, temperature, etc.) and log them*
4. Detect collisions of particles with each other*
5. Apply, if neccessary, merge reactions*
6. Apply, if neccessary, transform reactions*
7. Calculate collision response for the remaining collision pairs*
8. Apply, if neccessary, decay reactions*
9. Paint the scene

You can provide alternate implementations of the steps marked with an asterisk*. Code the alternate implementation as a Java class and instruct ‘saces’ to use it. For a complete treatment of not only the simulation loop, but the whole simulation process, see section 6.3.2, page 37.

3.4.3 The data viewer

Thermodynamical data like temperature and pressure, and mechanics like particle count, a histogram of particle speeds and other data are presented as diagrams in the data viewer (see figure 3.3).

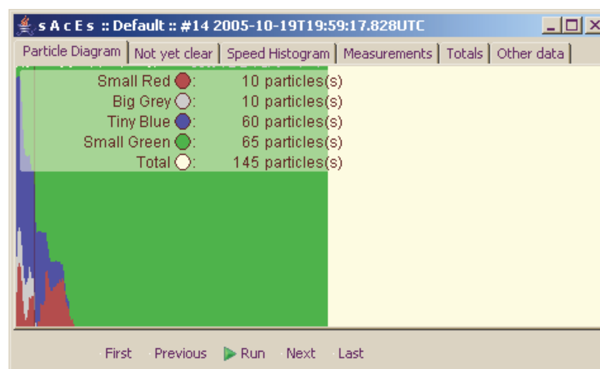


Figure 3.3: Todo: The particles diagram of the Lotka-Volterra experiment

The data viewer can be started as a separate process, or even on another workstation if the binary log file is stored on the network.

3.4.4 The experiment settings dialog

There are general experiment settings like initial temperature or test tube dimensions (see figure 3.4), then there are particle settings to define particle properties and reaction settings to define reaction equations and finally experiment properties as key-value pairs for additional experiment parameters and to instruct ‘saces’ to use alternate implementations of simulation steps.

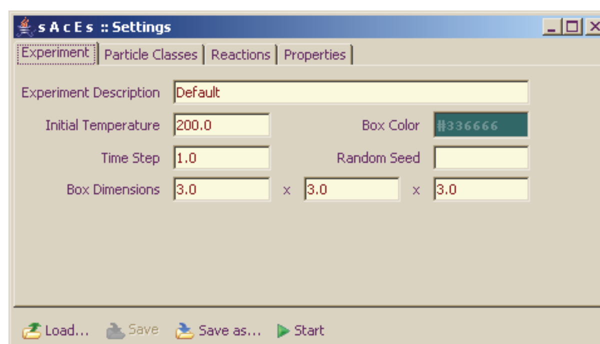


Figure 3.4: The Experiment settings window

3.4.5 Saving simulation data

Because the simulation is a CPU-, memory- and graphics-intensive application, communication is kept to a minimum. The tool and the simulation loop use a

binary logger to save simulation data to a file. The file can be placed on a network share and be read by a second instance of ‘saces’ on another workstation to display diagrams or raw data.

3.5 Getting started with ‘saces’

3.5.1 Getting ‘saces’

‘saces’ is delivered as a *.jar* file for all platforms. Get it from the diploma CD or from the project homepage `http://saces.yce.ch`.

For Windows use the executable `saces.exe`, and for Mac OS X 10.4 the application bundle `saces.app`. (todo: create the executable and application bundle!)

3.5.2 Java Runtime Environment version 5

‘saces’ works with Java version 5 and up only. You need to install at leaset the Java Runtime Environment version 5. Download it from `http://java.com/en/download/manual.jsp` and follow the installation instructions.

3.5.3 Optional: Getting JOGL

Note! This step is not necessary with the Windows or Mac OS X executables. Do this if you use the *.jar* file. ‘saces’ needs JOGL native libraries. They are already included in the platform executables, but not with `saces.jar`.

1. Go to `https://jogl.dev.java.net` and search your platform on the download page or get the libraries from the diploma CD.
2. Download the `jogl-natives-`<*platform*>`.jar` file.
3. For Windows unpack with *WinZip* and copy the *.dll* files to the same directory as the `saces.jar` file or in the *ext* subdirectory of the Java installation. For JOGL release 1.1 they are `jogl.dll` and `jogl_cg.dll`.

   For other platforms proceed similarly.

3.5.4 Start 'saces'

No need for installation. Just double-click the icon.

A window with a black empty background is showed. This is the main view. Click on the 'Run' button at the bottom. A sample experiment is started. Molecules are displayed as colorful spheres with different sizes. OpenGL lighting is applied to give a more realistic view. The reaction vessel is painted, too. Some walls left out to allow a better view into the reaction space (see figure 3.2, page 19).

3.5.5 Stop simulation and load experiment

Just hit the 'Settings…' button while the simulation is running. This stops the simulation and shows the settings window. Then click the 'Load…' button and load the `volterra.xml` experiment specification file. The experiment specification file is on the diploma CD or can be downloaded from the project homepage `http://saces.yce.ch`.

3.5.6 Tweak the experiment

Try to change the experiment tube wall colors to a bright red (see figure 3.4). Then change the initial count of preys to 2500. Click on the 'Particle Classes' tab and select the 'Initial Count' column of the row with particle name 'Prey' and edit the number. If you like change its color, too. (see figure 3.5).

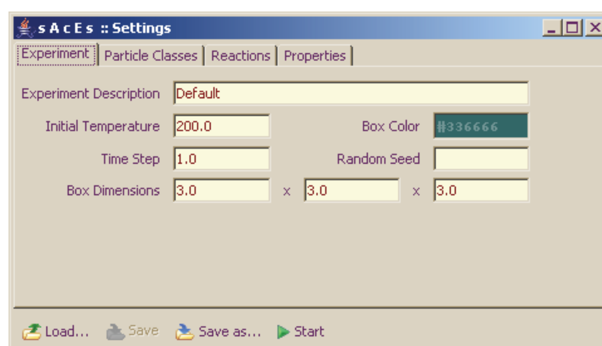


Figure 3.5: Todo: The Particle classes settings tab

Then try to change the probability of catching prey. Click the 'Reactions' tab and edit the probability of the 'Catch Prey' reaction to 0.75.

3.5.7 Start the new experiment

You can save the modified experiment or start the simulation right now. Click on the ‘Run’ button. The settings window disappears and the new experiment is running.

3.5.8 See the binary log

While the new experiment is running, we open a shell (DOS Prompt on Windows, Terminal on Mac OS X) and enter:

```
java -cp saces.jar saces.file.BinaryLogRetriever
volterra.binlog volterra.xml
```

This dumps the contents of the binary log in text form. `BinaryLogRetriever` works like `tail -f` in Unix. That means, the process doesn't end if it reaches the end of the binary file, but waits for more data from the running simulation. You have to cancel it with Control-C if you are satisfied.

If you prefer outright visual data, click on the ‘View Data…’ button, and you have a selection of diagrams (see figure 3.3, page 20).

3.5.9 The snapshot

If you want to know “everything”, click on the ‘Snapshot…’ button. This saves a snapshot to the binary log file. The snapshot contains velocity, position and particle class of all particles in the simulation the moment the button has been clicked. Convert the snapshot to textual data for further processing with `BinaryLogRetriever`.

Chapter 4

The User Manual

4.1 Overview

4.1.1 The main view

4.1.2 The settings window

The general settings tab

The particle classes tab

The reactions tab

The properties tab

The 'Properties' tab allows to change the experiment properties. The experiment properties are a collection of name-value pairs: each property has a name, the key, and a value which is a string or a number. With properties one can define additional settings in an extensible way.

The 'Properties' tab looks like this (figure 4.1). Double-click on a cell to edit its value. A new property is added with the click on the 'New Property' button. A new empty row is created and the new name and value can be entered. A property is deleted with the click on the 'Delete Current Property' button if a row has been selected before. It is not possible to create two properties with the same name.

The simulation loop plug-and-play architecture uses the experiment properties to define the steps in the simulation loop. The steps are defined as dynamically loadable Java classes implementing specific interfaces. The reflection of the

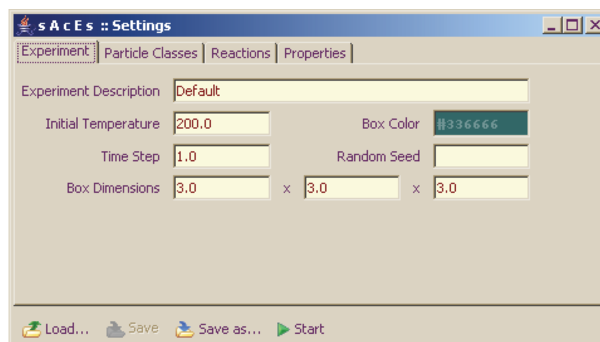


Figure 4.1: Todo: The Properties settings tab

particles at the reaction vessel walls, for example, is defined as a property with the name `WallReflector`. The value of the property is the name of the Java class implementing the reflection. We can replace this class to include heating or cooling, for example. Such a class `yourpackage.HeatingReflector` must implement the interface `saces.pnp.WallReflector` with its single method:

```
public void reflect(
    Particle[] particles, Simulation simulation);
```

It is possible to pass parameters to the extension classes. Just define a property with a sensible name and put the parameter as a value to the property. Sensible names could be a catenation of the extension class name and the parameter name, separated by a dot. The class `saces.pnp.DistributorRandom`, for example takes the parameter `DistributorRandom.stdDev` for the standard deviation of each of the three velocity components.

For a table of properties, please see table 4.1.

4.2 Navigation of the three-dimensional view

4.3 Plug-and-play architecture

4.4 Compiling 'saces'

(todo: technical limitations like the maximum 10'000 particles.)

Table 4.1: The property keys and their meaning

Key	Default	Meaning
BinaryLog	saces.binlog	The file name of the binary log file.
HistogramTime	1000	The time in milliseconds between histograms.
HistogramSize	100	The number of histogram steps (the resolution of the histogram).
HistogramMax	0	The maximum value of the histogram. If it is 0 (zero), use the maximum speed encountered so far in the simulation.
Particle	saces.gl.Sphere	Name of Java class to implement a particle. The class must be able to render the particle in OpenGL.
Distributor	saces.pnp.Distributor-MaxwellBoltzmann	Name of Java class to provide the initial distribution of the particles in an experiment. The class must implement the interface saces.pnp.Distributor.
DistributorRandom.stdDev	1	A parameter for the class saces.pnp.DistributorRandom, namely the standard deviation for the speeds distributed to the particles.
Measurer	saces.pnp.MeasurerDefault	Name of Java class to provide the measurements of temperature, pressure, etc. The class must implement the interface saces.pnp.Measurer.
MeasurerTime	1000	The time in milliseconds between measurements. (todo: implement)
Reflector	saces.pnp.ReflectorSimple	Name of Java class to provide the reflection of particles at the reaction vessel walls. The class must implement the interface saces.pnp.Reflector.

Table 4.1: continuing…

Table 4.1: continued

Key	Default	Meaning
Detector	saces.pnp.Detector-Partitionized	Name of Java class to provide the collision detection. The class must implement the interface saces.pnp.Detector.
Detector-Partitionized.particleCount-PerPartition	12	A hint to the partitionizer about how many partitions are to create. The value is the average count of particles in each partition (see section 6.5 page 46).
Merger	saces.pnp.MergerSimple	Name of Java class to handle merge reactions. The class must implement the interface saces.pnp.Merger.
Transformer	saces.pnp.Transformer-Simple	Name of Java class to handle transform reactions. The class must implement the interface saces.pnp.Transformer.
Response	saces.pnp.ResponseSchwab	Name of Java class to provide the collision response. The class must implement the interface saces.pnp.Response.
Decayer	saces.pnp.DecayerSimple	Name of Java class to handle decay reactions. The class must implement the interface saces.pnp.Decayer.

Table 4.1: The end

Chapter 5

Some Experiments with ‘saces’

todo: two or three experiments with ‘saces’, one of them should be truly “Artificial Chemistry”, like the Lotka-Volterra systems

5.1 The sample experiment

todo: some notes about “selfish classes”?

With the start of ‘saces’ a sample experiment is pre-loaded and immediately ready to be started. This is very helpful for first-time users because they can start “playing” immediately and get a first feel of the simulation at once. (This is the “Batteries Included” philosophy: The user doesn’t need to do additional things like go buying missing batteries to get a first positive experience: It works out of the box!)

The sample experiment is a very simple experiment with four particles of colors red, green, blue, and a greyish white. The blue particles are tiny and abundant, while the grey particles are big and rare.

todo: write more about the sample experiment.

5.2 A chemical explosion

5.3 A Lotka-Volterra system

5.4 Brownian movement

5.5 A finite automaton

Chapter 6

The Internals of the tool 'saces'

6.1 Internal structure

(todo: write about the packages, saces.app, etc.) AND [todo: repackage thusly

saces.app

saces.app.gui

saces.file

saces.gl

saces.exp (Experiment, ParticleClass, Reaction)

saces.sim (Mediator, Simulation, Particle, Partition, ...)

saces.pnp

and the steps:

1. application → app
2. particle *remove* (ParticleClass → exp, other → sim)
3. simulation → sim

]

6.2 Data architecture

In ‘saces’ a distinction between experiment setup parameters and simulation state is made. Former data is constant (usually one doesn’t change rules in the middle of a simulation), latter data is dynamic. And another difference is the ability to load and save experiment setups. Simulation state can’t be loaded. (Perhaps this is a future feature of ‘saces?)

It is possible to save a snapshot of simulation state, but this snapshot cannot be reloaded again into the simulation.

6.2.1 Experiment setup

The experiment parameters consist of three classes. They are class *Experiment*, class *ParticleClass* and class *Reaction* (see figure 6.1).

Class *Experiment*

An *Experiment* instance provides global experiment parameters (like *initial* temperature in the reaction vessel). And it is used as a starting point to access the experiment data. There are particle class and reaction lists. There is only one instance per running simulation. Of course a running simulation can have only one setup.

Class *ParticleClass*

A particle class defines a particle “species”. Particles of same species all have same mass, for example. Mass and other parameters are stored in a *ParticleClass* instance.

The term “class” can be misunderstood by Java developers. A class is something like a blueprint that defines common characteristics of objects. This is true for particles as well. A particle class is not a class for Java instances. It is a definition of common parameters for particles.

Each particle class has both an unique name and an index. We find *ParticleClass* instances starting from the *Experiment* instance using either the name or the index.

In the simulation loop it must be found efficiently which reactions a particle can do. So a particle class contains three reaction lists, one for each reaction type. These lists contain the reactions of which the particle can be an educt.

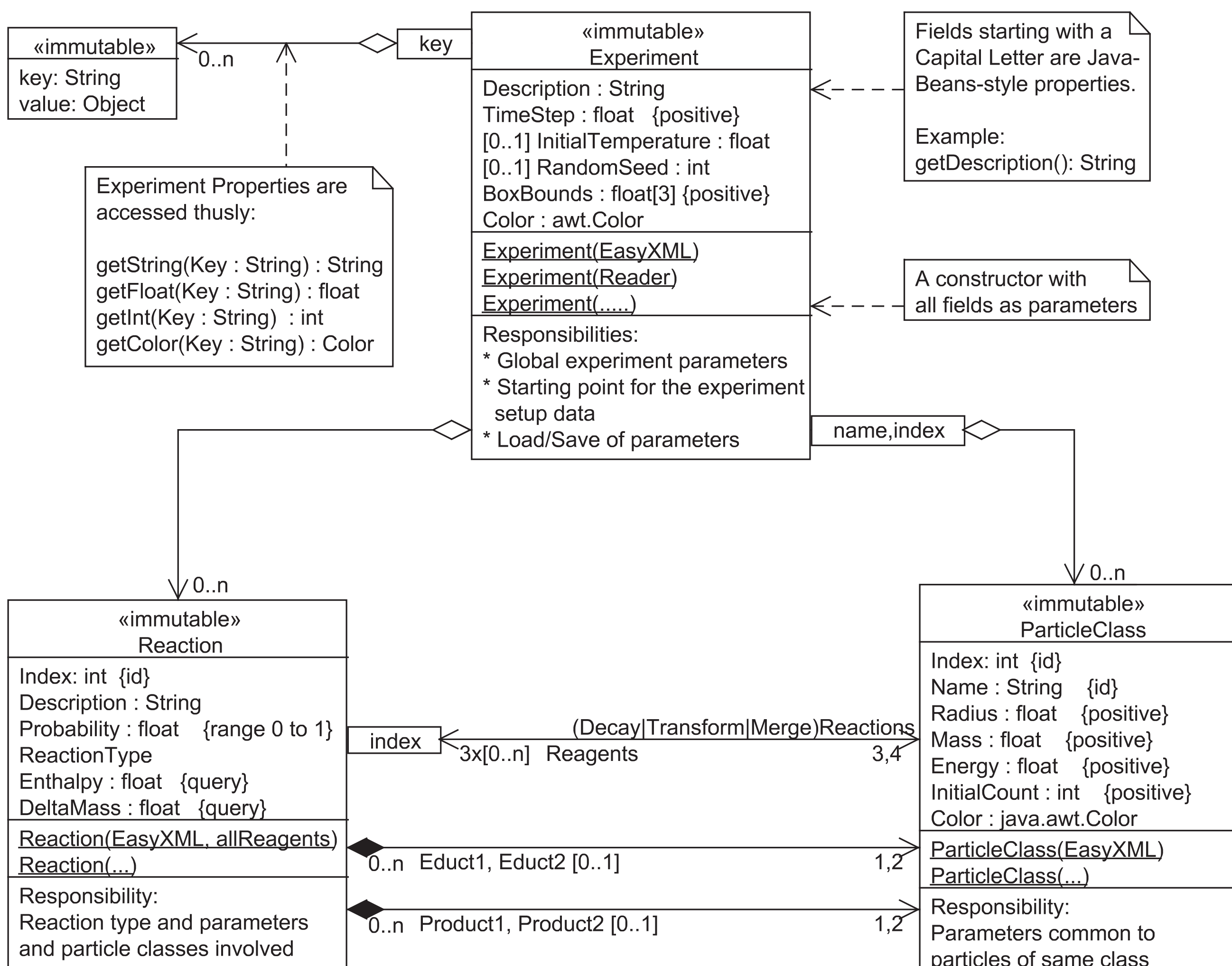


Figure 6.1: UML class diagram for the experiment setup

Class *Reaction*

A *Reaction* instance defines a reaction in ‘saces’. It contains reaction parameters: the reagents, activation energy, probability of reaction, etc.

There is a reagent list with three or four elements (depending on reaction type). The order of elements in the list is important. First the educts are listed, then the products.

Alternatively, access the *Educt1*, *Educt2*, *Product1* and *Product2* Java-Beans-style properties directly. The second educt or product are optional, of course. For *Merge* reactions *Product2* is not defined, and for *Decay* reactions *Educt2* is not defined. That means the getter returns a null pointer.

Reaction Type

There are three reaction types (see section 3.2.1, page 16): *Transform*, *Merge* and *Decay*. The *ReactionType* Java-Beans-style property can have one of these three values. With Java 5, enumerated types are possible. *ReactionType* is an enumerated type with these three values.

Experiment Properties

Additional parameters are accessed as key-value pairs (see table 4.1, page 26). Experiment properties are important to specify additional data for the experiment in a flexible way.

6.2.2 Simulation State

A side note

Many design decisions explained in this section were influenced by ‘performance’. We didn’t do so-called ‘premature’ optimization. We optimized only if we discovered evidence of a performance problem. There were two ways to discover such problems. The most general was complexity theory. For the naive collision detection we calculated a time complexity of $O(n^2)$, for example. The second one was just measuring. With a good Java profiler we analyzed running times and invocation count per class and per method.

Simulation State

The experiment setup defines static, initial parameters and data for the experiment. Simulation state, however, is much more dynamic. It contains, for example, the velocity and position of all particles, and they change very much during the simulation, of course.

The two most important classes are class *Simulation* and class *Particle* (see figure 6.2). Then, for an efficient collision detection Space Partitioning is important. It is handled by the class *Partition*. (How space partitioning works, see section 6.5, page 46.)

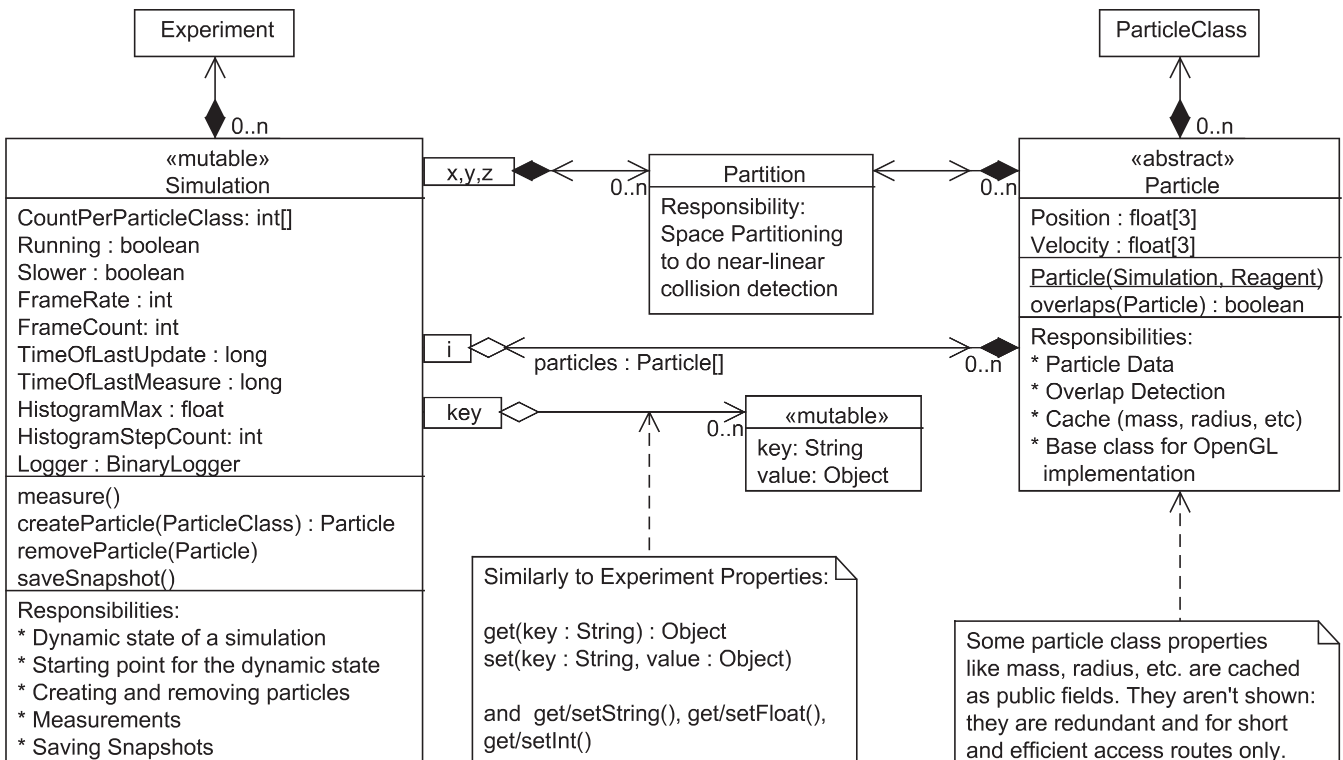


Figure 6.2: UML class diagram for the simulation state

The classes contain many hot spots. A very hot spot is the method *Particle.overlaps()*. In a very naive (quadratic) implementation of collision detection of ten thousand particles it is called almost fifty million times each time. At 20 frames per second, this would be a billion (10^9) times per second.

With partitioning it is called about sixty thousand times, which is still very 'hot'. That's why we have public member fields in the classes. This is not con-

forming to our Java Coding Convention [Amb00]. We just can't afford Java-Beans-style getter method calls, however.

Class *Simulation*

A *Simulation* instance contains dynamic state (like current temperature, the particle counts per particle class, etc) and is used as a starting point to access the different parts of simulation state. There is a list of particles. The three-dimensional array of partitions (for space partitioning) is found in the *Simulation* class as well.

The *Simulation* instance is responsible for the creation and removal of particles, too. This is neccessary to keep the particle counts per particle class up to date. Measurements like temperature, frame rate, etc. are done as well.

The simulation loop is not done here. It is done by the *Mediator*.

Class *Particle*

The particle is one of the foci of our interest in the simulation. Where is the particle? And how fast does it move in the reaction vessel? Particles are very dynamic: they move fast and change velocity often, they increase and decrease and even disappear completely if a reaction has consumed up a species. The only thing which stays the same is its class or species, respectively.

There is the method *overlaps()* which tests whether the particle overlaps with another particle. This method is the hottest spot of the whole tool — it is so hot that it has been considered whether it should have better been inlined.

Because *overlaps()* needs the radius of the particles, it is cached as a public final member field. When a particle is created, the radius is copied from the particle class and left unchanged till the end of the particle. Other particle class parameters are cached as well in this way.

The class *Particle* is abstract. The *Sphere* class extends it and provides an implementation as an OpenGL sphere. So there is no *Particle* constructor. It is the *Simulation* instance that creates and removes particles. An experiment property decides which implementation class to use (see key `Particle` in table 4.1, page 26).

Class *Partition*

For more details see section 6.5, page 46. Here it suffice to say that both *Simulation* and *Particle* instance maintain bidirectional references to *Partition* instances.

This induces tight coupling. Such tight coupling should be avoided in a good software design. Here again we have performance problems, however. Whenever a moving particle crosses a partition boundary, some handling with particle lists is neccessary. This happens quite often. Short access routes in two directions are neccessary, see method *CollisionDetectorPartitionized.repartitionize()*. But, again, for details see section 6.5.

Simulation Properties

Similar to experiment properties there are simulation properties, but they are changeable. They are needed to keep and pass around additional data from a step to another. For example the number of reaction of a type is kept in a simulation property and used for measurements.

(todo? a table of simulation properties?)

Other Properties

There is a need for properties per particle, collision pair and partition. Collision detection should be done only once for each particle. Then particles must be marked as already detected, for example. Collision pairs are augmented with reaction enthalpy and mass exchange for the collision response. Most of these properties are hard-coded as fields. It is not sure whether key-value pairs are needed in the long term. They are not included in the proposed ‘saces’.

(todo: verify this in the source code)

6.3 The process architecture

6.3.1 The mediator

The mediator mediates between the GUI and the state of both OpenGL and simulation. For example, if the user clicks on the ‘Run’ button, the button event handler calls *Mediator.setRunning(true)*. The GUI doesn’t interact directly with OpenGL or the simulation state. User requests are mediated and forwarded where they make sense. The mediator is the only class importing JOGL classes not in the *gl* package of ‘saces’. And classes in the *app* package don’t use any classes in the *sim* package except the mediator.

Another responsibility is forwarding the JOGL display event to the simulation loop. The JOGL framework uses an event-driven architecture. ‘saces’ doesn’t decide *when* to paint, it is JOGL who decides.

This works thusly: The mediator implements the interface *GLEventListener* of JOGL. The interface contains the method *display()*. Whenever OpenGL is ready and wants to display a new frame that method is called. The mediator then forwards the calls to the simulation loop. (And if the simulation is halted just the painting is done.)

6.3.2 The simulation process

The most important part of a simulation is the simulation loop. A simulation is an endless loop where the simulation state is being calculated repeatedly. The loop runs as long as the simulation is left running (see figure 6.3).

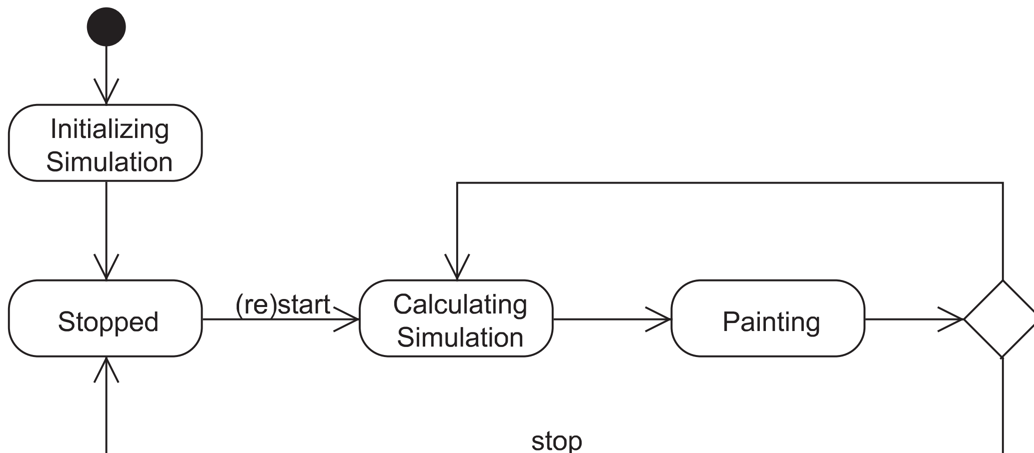


Figure 6.3: UML action diagram: a simple simulation process

This is a very simple example of a simulation process. ‘saces’ works exactly this way. After initialization, the simulation doesn’t start immediately. The user starts the simulation. Calculation and painting is done repeatedly till the user stops the simulation.

The states *Inititalizing* and *Calculating Simulation* can be subdivided in more complex sub-processes. (see figure 6.4). The following sections explain these sub-steps one by one. The sub-steps printed in cursive are available to the plug-and-play simulation architecture. For example, the advanced user might provide a different initial distribution of the particles other than the default Maxwell-Boltzmann one.

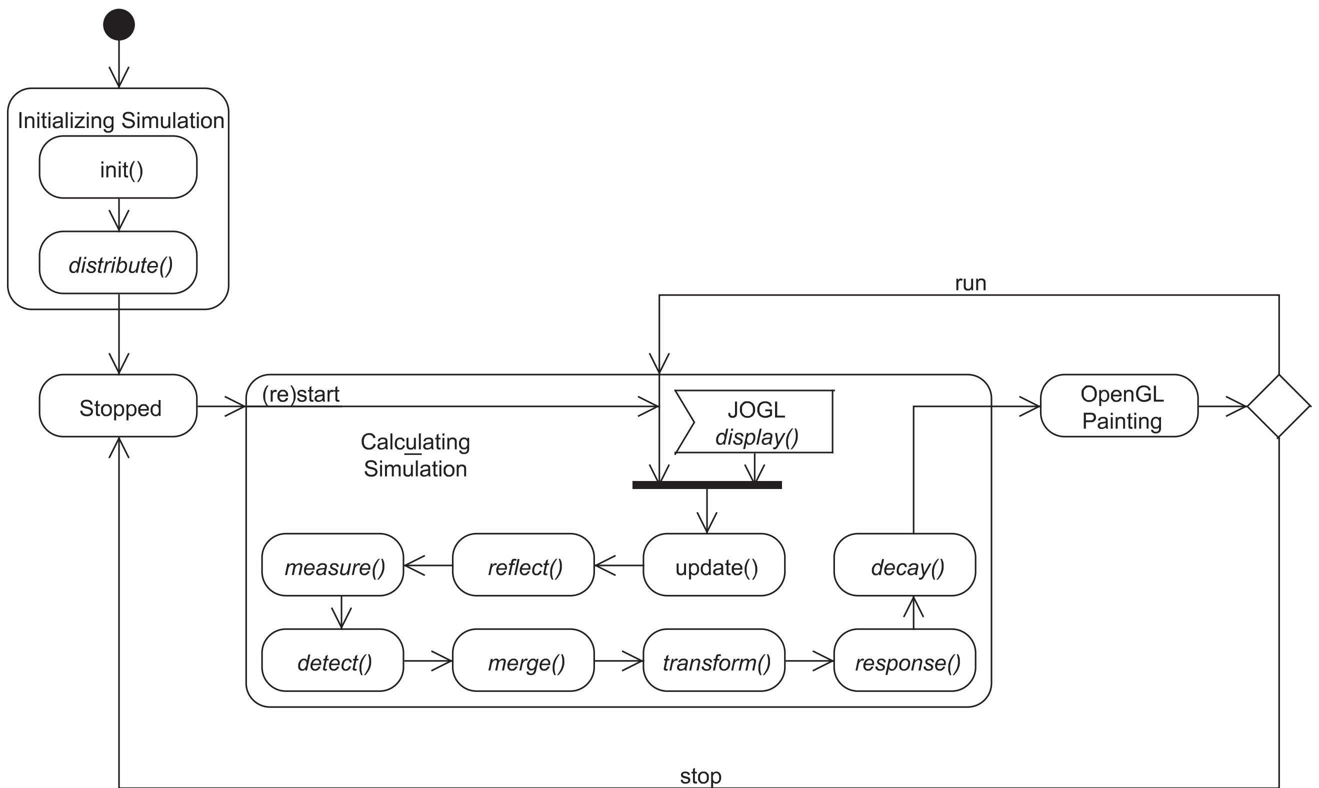


Figure 6.4: UML action diagram: a more complex simulation process

The steps can save data to the binary logger. Especially *measure()* writes important data. For more details about the binary logger, see section 6.3.3, page 46.

Another aspect is the handling of data flow between the steps. *Transform* reactions need a list of collided particles from the collision detection. *Merge* reactions remove merged pairs from this list, etc. To pass collision pairs an array of *Collision* instances are used. They can store additional properties like enthalpy for the step *response()*.

Our first approach in development of 'saces' tried to use the *Visitor* pattern. A particle was visited by a *TransformReactionVisitor* instance if it was to be used in a *Transform* reaction, for example. We discovered very soon that the pattern was not useful. The method calling overhead was too much. We cannot afford to call a method for every single particle that underwent a reaction. After we switched to a "do-it-all-at-once" approach with arrays performance soared. All *Transform* reactions are handled with *one* call of the *transform()* method of the interface *Transformer* in one sweep, for example. Figure 6.5 shows a data-flow diagram.

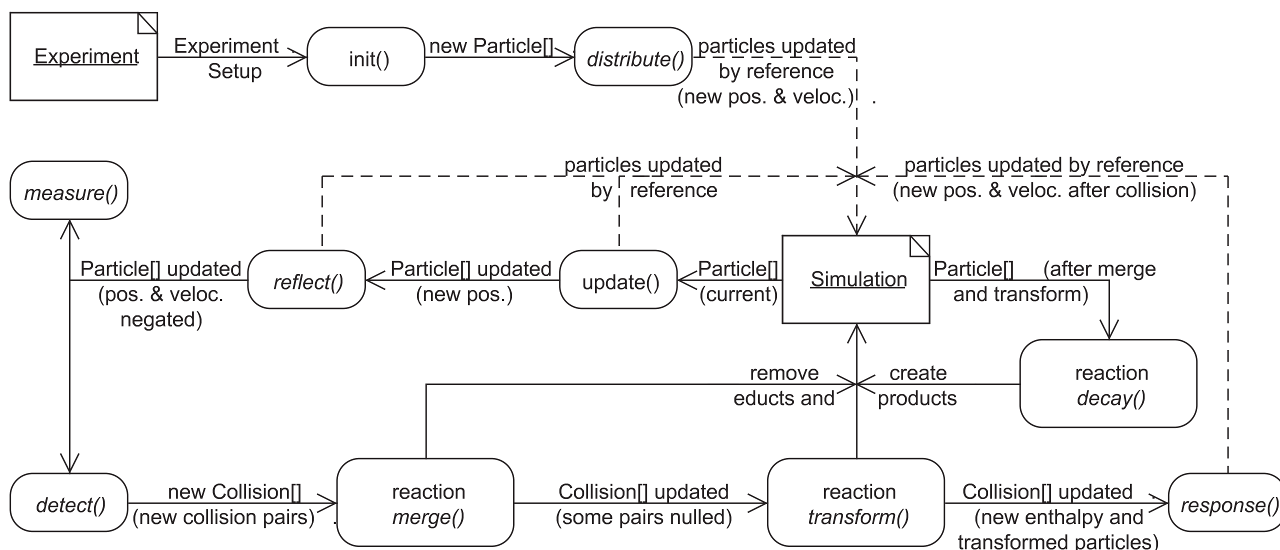


Figure 6.5: Data flow diagram: how data flows in the simulation process

Particle arrays and collision pair arrays are created and passed around instead of invoking methods for every single particle. Steps in cursive are available for the plug-and-play architecture. The data flows are:

Particle arrays are passed to *distribute()*, *update()*, *reflect()*, *measure()* and *detect()*. They are passed to *decay()*, as well, but as last step in the simulation loop.

Updating particles by reference is done by the steps *distribute()*, *update()*, *reflect()* and *response()*.

Removing educts and creating products is done by the reaction steps (a bit bigger) *merge()*, *transform()* and *decay()*.

Collision pairs are created by *detect()* and passed through to *merge()*, *transform()* and finally *response()*.

Modifying collision pairs is done by *merge()*, which removes some pairs, and by *transform()*, which augments the pair with additional information and replaces the particles of the pair.

Simulation properties can be read and written by all steps.

Binary log is available for all steps to write data, especially *measure()* needs the binary log.

Initializing data flow is implicit through the mediator instance. Used by *init()*.

update(), for example takes the current particles as an array and modifies particles by reference. And *measure()* doesn't modify anything except that it writes to the binary log and perhaps saves data to the simulation properties. The reaction steps *merge()* and *transform()* take collision pair arrays, amd *decay()* a particle array. They remove educt particles and create product particles directly in the simulation state.

For the exact signatures of all simulation process steps and the ins and outs of them, see table 6.1. The table contains the interface names as well without the package name *saces.pnp*. The table only shows data flow by parameters, return values and direct adding or removing of particles *(simulation out)*. A special case is *init()*, because data flow is implicit through the mediator instance (neccessary to initialize first).

(code todo: put all default implementations into the mediator because they are useful for pnp)

Description of the steps

The next sections contain the detailed description of the simulation process steps starting with *init()* and ending with *decay()*.

Table 6.1: Data flow and step signatures of the simulation process

Step	Interface	Signature and Data flow
init()	part of the mediator	`void init()` *implicit:* simulation from mediator *implicit:* new particle array is created in simulation
distribute()	*Distributor*	`void distribute(Particle[], Simulation)` *in:* particle array and simulation *out:* array with *updated* particles ($\vec{p}, \vec{v}$)
update()	*Updater* (in package *sacessim*)	`void update(Particle[], Simulation)` *in:* particle array and simulation *out:* array with *updated* particles ($\vec{p}$)
reflect()	*Reflector*	`void reflect(Particle[], Simulation)` *in:* particle array and simulation *out:* array with some *updated* particles ($\vec{p}, \vec{v}$ negated)
measure()	*Measurer*	`void measure(Particle[], Simulation)` *in:* particle array and simulation
detect()	*Detector*	`Collision[] detect(Particle[], Simulation)` *in:* particle array and simulation *return:* collision pairs array
merge()	*Merger*	`void merge(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with *deleted (null)* elements *simulation out: removed* and *new* particles
transform()	*Transformer*	`void transform(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with properties *added* (like enthalpy) and where particles have *changed* particle class *simulation out: removed* and *new* particles
response()	*Response*	`void response(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with same elements but where particles have *changed* velocity (and perhaps position)
decay()	*Decayer*	`void decay(Particle[], Simulation)` *in:* particle array and simulation *simulation out: removed* and *new* particles

Creating particles—*init()*

This and the next step *distribute()* are done immediately after loading an experiment. This simplifies the state diagram, because the *Stopped* state always means that the experiment is initialized and ready to be started or restarted.

So, because the program is always started with a default experiment (which is saved within and as part of the code), *init()* and *distribute()* are run the first time even before the user did anything. This shouldn’t pose problems because the default experiment is designed carefully not to produce problems and is unchangeable within the *.jar* file.

Creating particles is quite a simple step. The initial counts of the particle classes determine how many particles are created. The particles are registered in OpenGL display lists for later display. Particle counts are set, too.

Then the plug-and-play classes are loaded and an instance to implement the steps created, as well. Reflection is used for the dynamic loading of Java classes and creating instances. If there is a problem with the reflection, an user-friendly error message is shown. For possible problems see (see section 6.6, page 46).

The method *init()* is part of the mediator and not available for enhancement as a plug-and-play step.

Initial Distribution of the particles—*distribute()*

After the particles have been created, they should be assigned position and velocity. The interface *saces.pnp.Distributor* contains one method *distribute()*. The distributor uses the particle array passed to the method to set the position and velocity of the particles and can use the *Simulation* parameter to look up, for example, the experiment setup for the initial temperature.

An example is the *DistributorRandom* class, which distributes the position uniformly and the velocity components normally with standard deviation saved in the experiment property `DistributorRandom.stdDev`.

Repositioning of the particles—*update()*

After each run of the simulation loop some real time has passed. The particles are at their new positions. Using velocities the new positions are calculated:

$$\vec{p}' = \vec{p} + \Delta t \cdot \vec{v}$$

This is an extremely simple step and we saw no need to provide a possibility to enhance *update()*. Therefore this step is one of the two steps not available to the plug-and-play architecture.

Reflecting at the reaction vessel walls–*reflect()*

(todo: swap reflect() and measure() in the code!!)

update() did not worry whether the particles left the reaction vessel boundaries. This is the task of *reflect()*. It negates per component both velocity v_σ and position p_σ if the position component is outside the range $[r \dots b_\sigma - r]$ where σ is either x, y or z. Particle radius r is considered.

$$v_\sigma \leftarrow -v_\sigma; p_\sigma \leftarrow -p_\sigma + \begin{cases} 2r & \text{if } p_\sigma - r < 0 \\ 2b_\sigma - 2r & \text{if } p_\sigma + r > b_\sigma \end{cases}$$

This is a simple step as well. With changing the sign even the effect that the particle already has traveled a tiny distance beyond the boundary is considered and the new position is set retroactively. This works because it is assumed that the boundaries are rectangular and orthogonal to each other (see section 3.2.1).

A possible improvement of this step is heating or cooling of the vessel walls—and this is why *reflect()* is pluggable.

Measurements of physical parameters—*measure()*

To learn about the world one uses the sense organs to perceive things. Physicists use measurement instruments to measure things. In ‘saces’ we do measurements in the *measure()* step. Measured are maximum, minimal and average speed of the particles, the mass center and total mass, bound energy and momentum of all particles, temperature and pressure.

(todo: explain how to measure temperature)

Measurements are written to the binary log. Because measurements need time they shouldn’t be done in every iteration. The experiment property `MeasurerTime` defines the time between measurements in milliseconds.

Collision detection

Conservation principles

‘saces’ doesn’t do mass conservation. It is up to the user’s responsibility to verify whether all reaction follow mass conservation. (todo: perhaps a button to verify mass conservation? and todo: write about energy and momentum conservation)

Merge reactions

Total inelastic collision (todo: write about energy conservation)

Transform reactions

Collision response

Introduction

Elastic collision

Collision with enthalpy

A solution (todo: write about solving the collision problem)

Approach of Dr. Hinze

Approach of Dr. Schwab

Summary

Decay reactions

Particles can decay with specified decay reaction probabilities. If there are more than one decay reaction, the sum of the probabilities shouldn’t exceed 1. It doesn’t make sense to normalize the probabilities because the probability of *not* decaying would be needed then.

Activation energy is not considered in a decay reaction. But a decay reaction cannot be activated if there is not enough bound energy:

$$H_e \geq H_{p_1} + H_{p_2}$$

The bound energy of the products should not exceed the educt bound energy H_e. Excess energy is transformed into kinetic energy. That means the decayed particles are faster than the educt particles (assuming mass is same before and after).

The collision response approach of Dr. Schwab can be reused here. The educts will be scattered randomly but momentum and energy will be preserved. The steps:

1. Create a collision pair and the educts
2. Set the educt positions and velocity to the ones of the product
3. Set mass exchange to zero.
4. Calculate eaction enthalpy $\Delta H = H_e - H_{p_1} + H_{p_2}$.
5. Call *response()*.

Other approaches need to calculate the scattering angles differently and cannot pass the pair to the collision response. That is why *decay()* is the last step of calculating the simulation. *DecayerSimple*, however, does leverage *CollisionResponseSchwab* by composition. A private *CollisionResponseSchwab* member variable is used.

Painting the scene with OpenGL

6.3.3 The binary logger

6.4 Handling Time

6.5 An optimization: Space Partitioning

(todo: include some measurements to show the time complexity of the space partitioning algorithm)

about binary space partitioning see [N+95]

6.6 How to implement the plug-and-play classes

Possible problems are

ClassNotFoundException if the Java classloader didn’t find the class. The user must set the classpath or move the class to the same directory as the *.jar* file. If the class has package, the class file must be put into the corresponding tree of subdirectories as Java requires it.

ClassCastException if the instance couldn’t be casted to the given interface. The class must implement the corresponding interface. For example to implement the *detect()* step, write a class which implements the iterface *saces.pnp.CollisionDetector*.

ExceptionInInitializerError if during class initialization an exception has been thrown. Try to write a test program which creates the instance and see what happens.

InstantiationException if the class is abstract or an interface. The class must be a concrete class.

IllegalAccessException if the constructor is not public. There must be a public constructor without parameters.

Linkage errors (rare) if the class file has a wrong format, could not be verified and other linkage problems.

No property key (rare: defaulting of properties failed) if there is no such experiment property. Try to define the property with the correct key.

6.7 OpenGL

6.8 Todo: Other sections?

List of Figures

3.1 The shape of the reaction vessel in ‘saces’ 16
3.2 Running the sample experiment 19
3.3 Todo: The particles diagram of the Lotka-Volterra experiment . . 20
3.4 The Experiment settings window 20
3.5 Todo: The Particle classes settings tab 22

4.1 Todo: The Properties settings tab 25

6.1 UML class diagram for the experiment setup 32
6.2 UML class diagram for the simulation state 34
6.3 UML action diagram: a simple simulation process 37
6.4 UML action diagram: a more complex simulation process 38
6.5 Data flow diagram: how data flows in the simulation process . . . 39

List of Tables

4.1 The property keys and their meaning 26

6.1 Data flow and step signatures of the simulation process 41

Bibliography

[Adl94] Leonard M. Adleman. Molecular computation of solution to combinatorial problems. *Science*, pages 1021–1024, 1994.

[Amb00] Scott W. Ambler. Writing robust java code: The AmbySoft Inc. coding standards for java v17.01d. AmbySoft Inc., `http://www.ambysoft.com/javaCodingStandards.html`, 2000. White Paper for Java software developers.

[BFS02] Gil Benkö, Christoph Flamm, and Peter F. Stadler. A graph-based toy model of chemistry. Institut für Theoretische Chemie und Molekulare Strukturbiologie, Universität Wien, 2002.

[Coo03] Stephen Cook. The P versus NP problem. *Journal of the ACM*, 50/1:27–29, 2003.

[Dav04] Gene Davis. *Learning Java Bindings for OpenGL (JOGL)*. AuthorHouse, 2004.

[Dev02] Keith Devlin. *The Millenium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*. Basic Books, 2002.

[DZB01] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries—a review. *Artificial Life*, 7:225–275, 2001.

[Ega95] Greg Egan. *Permutation City*. Eos, 1995. Science fiction novel about an artificial life.

[FG04] Corina Frutschi and Pascal Grossniklaus. Simulation ausgewählter DNA-Computing Operationen. Enigneer's thesis, Hochschule für Technik und Informatik Bern, Abteilung Informatik, 2004.

[Fra03] Irmgard Frank. Chemische Reaktionen „on the fly“. *Angewandte Chemie*, 115:1607–1609, 2003.

[FS96] Martin Fowler and Kendall Scott. *UML konzentriert. Die neue Standard-Objektmodellierungssprache anwenden.* Addison-Wesley, 1996. Original title: *UML Distilled. Applying the Standard Object Modeling Language.*

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and Johnson Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hin05] Thomas Hinze. *Arbeitsmaterial zum Diplomprojekt Ein Experimentiersystem zur Simulation des Ablaufs chemischer Reaktionen*. Technische Universität Dresden, Institut für theoretische Informatik, Arbeitsgemeinschaft Rechnen mit Molekülen, 2005. Preliminary specification for the 'saces' tool.

[HS04] Thomas Hinze and Monika Sturm. *Rechnen mit DNA*. Oldenbourg Verlag München, 2004.

[Kne90] Fritz Kurt Kneubühl. *Repetitorium der Physik*. Teubner Studienbücher, 1990.

[Kuc96] Horst Kuchling. *Taschenbuch der Physik*. Fachbuchverlag Leipzig, 1996.

[LCF04] Rodrigo G. Luque, João L. D. Comba, and Carla M. D. S. Freitas. Broad-phase collision detection using semi-adjusting BSP-trees. Instituto de Informática, UFRGS, Brazil, 2004.

[LJ05] Remo Lehmann and Bojan Jambrešić. Simulation der Arbeitsweise eines DNA-Chips. Engineer's thesis, Hochschule für Technik und Informatik Bern, Abteilung Informatik, 2005.

[N+95] Bruce Naylor et al. A FAQ about binary space partitioning trees. `http://www.faqs.org/faqs/graphics/bsptree-faq`, 1995. FAQ.

[Nad02] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University of Jerusalem, 2002.

[NBH01] Andreas Ninck, Leo Bürki, and Roland Hungerbühler. *Systemik: Integrales Denken, Konzipieren und Realisieren*. Orell Füssli Verlag, 2001.

[SBB+00] Andre Skusa, Wolfgang Banzhaf, Jens Busch, Peter Dittrich, and Jens Ziegler. Künstliche Chemie. *Künstliche Intelligenz*, 1/00:12–19, 2000.

[Sce04] Eric R. Scerri. Just how ab initio is ab initio quantum chemistry? *Foundations of Chemistry*, 6:93–116, 2004.

[Sil05] Stephen Silver. Life lexicon, release 24. `http://www.argentum.freeserve.co.uk/lex_home.htm`, 2005. List of Game of Life terms; see term *Universal Computer*.

[Tom04] Kazuto Tominaga. Modelling DNA computation by an artificial chemistry based on pattern matching and recombination. Tokyo University of Technology, Hachioji, Tokyo, 2004.

[Win01] Herbert Windisch. *Thermodynamik*. Oldenbourg Verlag München, 2001.

[WKL+03] H.-J. Werner, P. J. Knowles, R. Lindh, M. Schütz, et al. Molpro, version 2002.6, a package of ab initio programs. Birmingham UK, `http://www.molpro.net`, 2003.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide Third Edition*. Addison-Wesley, 1999.

[You04] W. Anthony Young. Comparison of collision detection algorithms. University of Waterloo, Canada, 2004.