Investigating and Analysing Natural Properties Enabled by Systemic Computation within Nature-inspired Computer Models

by

Erwan Le Martelot

A thesis submitted in partial fulfillment for the degree of Doctor of Engineering

in the
Faculty of Engineering Sciences
Department of Electronic and Electrical Engineering

June 2010
Declaration of Authorship

I, Erwan Le Martelot, declare that this thesis titled Investigating and Analysing Natural Properties Enabled by Systemic Computation within Nature-inspired Computer Models and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at University College London.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at University College London or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date:
Whether considering a biological brain solving complex daily tasks, the growth of an organism according to its gene pool or the pigment patterns on some seashells, nature seems to be performing some kind of computation. Natural processes demonstrate many useful features that would be of great use for modern technologies. The capabilities of nature are such that biological systems have become highly significant to computer science.

However, conventional computation as enabled by modern computers operates according to very different properties than natural processes do. In that respect, systemic computation was introduced as a novel method of computation based on systemics, incorporating at its core natural computation properties not found in conventional computation paradigms and stressing the importance of structure and interaction within natural systems. Systemic computation also suggests a novel computer architecture compatible with current technologies, thus bringing together electronic and natural computations.

This thesis investigates and develops systemic computation by assessing its potential to enable exploitation and analysis of natural properties within nature-inspired computer models. It presents the first high-level complete systemic computer, including a visualisation framework, and models developed to study the properties of self-adaptation, fault-tolerance, crash-proof computing, self-repair, homoeostasis, flexibility and self-organisation. Each investigation is performed by creating a bio-inspired model in order to demonstrate how the rules and native natural properties provided by systemic computation can be exploited to build improved natural properties. Models include genetic algorithms, artificial neural networks, an artificial organism, a bistable gene network and a mitogen-activated protein kinase cascade. The work also discusses the advantages of the resulting nature-inspired computer models over conventional approaches.
Acknowledgements

Clearly, this thesis would not have been possible without the fantastic academic supervision of Peter Bentley and his initial idea of systemic computation. I am humbly grateful for all his dedicated guidance, encouragement throughout my doctorate and his tremendous help in the intimidating task of thesis write-up. I am also thankful for the good time shared during conference trips, outside of our scientific duty, exploring nice places and commenting on the beauty of nature. I also wish to particularly thank Beau Lotto for welcoming me in his lab and giving me the chance to start a doctorate I was so aspiring to do. I greatly appreciated his support, the freedom I got to conduct my research and the nice working environment he cultivated.

Further thanks go to Bob Sutherland for welcoming me on-board the EngD programme. I am very thankful to him as well as Paul McKenna for supporting my work and helping through any question or matter I could have regarding my doctorate. Their kindness, help and advices were deeply appreciated.

I also wish to express my sincere thanks to the people from the lab, institute and beyond that made my working environment such an insightful and fun place to be: Thanks to David Malkin, Udi Schlessinger and Martina Wicklein, the clan from the old lab on the 3rd floor for all the good moments, whether debating about science, life, or more material issues such as the arguable food quality in the canteen. Thanks to Daniel Hulme for helping me in the early steps of my work and for many interesting martial arts discussions. Thanks to Mark Miodownik for taking time in the early days of my work to help me with the physics of crystals. Thanks to Jamie Theobald for nice conversations and to him and Melissa for my first Thanksgiving! Thanks to Lefkothea Andreou, Katherine Bash, Peter Bex, Madi Boyd, Emma Byrne, Richard Clarke, David Corney, Annelie Maass, Adam Sillito and Umesh Telang for interesting conversations, nice coffee breaks or helpful advice. Last but not least, thanks to my mate Ilias Bergstrom for stimulating so many nice, fun or alternative conversations and making each working day a great lot of fun. In memory of all these good lunch food pilgrimages we shared, I also wish to share a thought for the nice people from the Whitecross street market, especially those from the now close to legendary burrito van and from the lovely bakery.

I would also like to thank Nick Burt, Nick Cameron, Sharon Edwards, Patricia Johnson and the staff from the institute of ophthalmology or from the electronic and electrical engineering department who provided help during my doctorate. I wish to acknowledge the EPSRC, the engineering department and the institute of ophthalmology for the financial support that made this work possible.
I am grateful to all the people, many of them being anonymous, who freely share their knowledge or resources on the internet in order to provide help for others, whether through Wikipedia, forums or their own websites. Thanks to Steve Gunn and Sunil Patel for sharing the TeX thesis template I reused and, in turn, modified. Thanks also to Robin Milner, Gianluca Tempesti and Yann Thoma for kindly giving me the authorisation to reuse some of their images.

I now wish to thank all the friends that made my London life so enjoyable and unforgettable. Big up to the crazy family for being this great unique nonsense melting-pot that led to memories sometimes only believable once facing the irrefutable evidence of photography. Special thanks to Iolino for being a supporting friend since I landed here. Köszönöm szépen to Koti for her support, unique mood management and the lovely time we shared. Thanks to Musti for so many happy stomach times. Lovely Sarah, thanks for being a lovely friend. Milene... ahh mia boba! Big thanks as well to my folks of the Camberley Soo Bahk Do club that made my Friday evenings so pleasant, sweaty, and a healthy way to relax after a week of scientific labour (Barry, you’re doing a great job!). Thank you all, home cuisine and odd bottle worshippers, cheap Japanese restaurants or pub’n poker addicts, whether from north or south of the river, from outer space or citizens of this world!

Surely none of this work would have happened without the support and love from my family to whom I am deeply grateful. A very special thank to my mom who unconditionally supported me throughout my life and gave me the chance to achieve what I wanted to. Warm thanks also to my sister and my brother in law for their help and support, to the new generation - my nieces and nephew - for being so kind and loving with their old uncle. Thanks to my mamie for still standing among us, always expressing her happiness when her grandson returns home for a few days, and never loosing her cheeky happy smile when spotting a box of chocolates. Thanks to the mom of Evie, our daughter, for being there for her.

Beyond what words can express, my thoughts go to my adored little daughter Evie who gives each beat of my heart a reason to be. She’s been since ever great and loving, and accepted my life in London in spite of the distance between us.

I’ll conclude these words by a vision of the Celtic west, and warmly greet the blue horizon in memory of a long time departed Breton, my dad...
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration of Authorship</td>
<td>2</td>
</tr>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>4</td>
</tr>
<tr>
<td>List of Figures</td>
<td>10</td>
</tr>
<tr>
<td>List of Tables</td>
<td>14</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>16</td>
</tr>
<tr>
<td>List of Listings</td>
<td>16</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>18</td>
</tr>
<tr>
<td>1.1 General Introduction</td>
<td>18</td>
</tr>
<tr>
<td>1.2 Systemic Computation</td>
<td>21</td>
</tr>
<tr>
<td>1.3 Hypothesis and Objectives</td>
<td>26</td>
</tr>
<tr>
<td>1.4 Contributions to Systemic Computation</td>
<td>28</td>
</tr>
<tr>
<td>1.5 Thesis Outline</td>
<td>29</td>
</tr>
<tr>
<td>1.6 Publications</td>
<td>30</td>
</tr>
<tr>
<td>2 Computational Biomimetism</td>
<td>32</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>32</td>
</tr>
<tr>
<td>2.2 Theory and Properties of Computation</td>
<td>34</td>
</tr>
<tr>
<td>2.2.1 Introduction</td>
<td>34</td>
</tr>
<tr>
<td>2.2.2 Foundations of Conventional Computation</td>
<td>34</td>
</tr>
<tr>
<td>2.2.3 Properties of Conventional Computation</td>
<td>41</td>
</tr>
<tr>
<td>2.2.4 Foundations of Natural Computation</td>
<td>44</td>
</tr>
<tr>
<td>2.2.5 Properties of Natural Computation</td>
<td>48</td>
</tr>
<tr>
<td>2.2.6 Summary</td>
<td>50</td>
</tr>
<tr>
<td>2.3 Hardware Approaches to Natural Computation</td>
<td>51</td>
</tr>
<tr>
<td>2.3.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>2.3.2 Conventional Hardware Architectures</td>
<td>51</td>
</tr>
<tr>
<td>2.3.3 Nature-inspired Hardware Approaches</td>
<td>55</td>
</tr>
<tr>
<td>2.3.4 Summary</td>
<td>60</td>
</tr>
</tbody>
</table>
2.4 Software Approaches to Natural Computation ........................................... 62
  2.4.1 Introduction ..................................................................................... 62
  2.4.2 Conventional Software Engineering Approaches .............................. 62
  2.4.3 Nature-inspired Software Approaches .............................................. 66
  2.4.4 Summary ......................................................................................... 85

2.5 Alternative Paradigms of Computation .................................................... 87
  2.5.1 Introduction ..................................................................................... 87
  2.5.2 Paradigms Originating in Mathematics, Physics or Technology .......... 87
  2.5.3 Paradigms Originating in Biology .................................................... 98
  2.5.4 Summary ......................................................................................... 108

3 Methodology ................................................................................................. 110
  3.1 Introduction ......................................................................................... 110
  3.2 Investigated Properties ........................................................................... 110
  3.3 Thesis Version of Systemic Computation .............................................. 112
  3.4 Systemic Computation Notations .......................................................... 114
    3.4.1 Calculus notation ........................................................................... 115
    3.4.2 Graphical notation ........................................................................ 117
    3.4.3 Comparison with the Stochastic $\pi$-calculus ................................. 120
  3.5 Systemic Computing ............................................................................ 122
  3.6 Systemic Analysis .................................................................................. 125
  3.7 Investigated Models .............................................................................. 127
  3.8 Analysis and Assessments ..................................................................... 129
  3.9 Summary .............................................................................................. 131

4 Platform ........................................................................................................ 132
  4.1 Introduction ......................................................................................... 132
  4.2 Virtual Machine ................................................................................... 132
    4.2.1 Systemic Computing Simulation ................................................... 133
    4.2.2 Program Loading and Memory Management ................................. 135
  4.3 SC Programming ................................................................................... 137
    4.3.1 SC Language ................................................................................. 137
    4.3.2 Defining Interactions ..................................................................... 151
  4.4 Platform Distribution ............................................................................ 153
  4.5 Summary .............................................................................................. 155

5 Self-adaptation within a Genetic Algorithm ................................................. 156
  5.1 Introduction .......................................................................................... 156
  5.2 Genetic Algorithm Model ..................................................................... 157
    5.2.1 Regular Model ............................................................................... 157
    5.2.2 Self-adaptive Model ...................................................................... 160
  5.3 Implementation of the TSP in a GA ...................................................... 161
    5.3.1 Presentation ................................................................................. 161
    5.3.2 Implementation ............................................................................ 162
  5.4 Experiments and Analysis ..................................................................... 166
    5.4.1 Experiments .................................................................................. 166
    5.4.2 Analysis ....................................................................................... 174
5.5 Summary ................................................................. 177

6 Fault-tolerance within a Genetic Algorithm ........................................... 178
  6.1 Introduction .................................................................. 178
  6.2 Testing Model .............................................................. 180
    6.2.1 Genetic Algorithm Model .......................................... 180
    6.2.2 Simulating faults .................................................... 182
    6.2.3 Implementing Self-maintenance .................................. 183
  6.3 Experiments and Analysis ................................................ 184
    6.3.1 Experiments .......................................................... 184
    6.3.2 Analysis ............................................................. 192
  6.4 Summary .................................................................... 193

7 Flexibility within Artificial Neural Structures ......................................... 196
  7.1 Introduction .................................................................. 196
  7.2 Artificial Neural Network Model .......................................... 197
    7.2.1 Systemic Analysis .................................................... 197
    7.2.2 Rules .................................................................... 201
  7.3 Experiments and Analysis ................................................ 206
    7.3.1 Experiments .......................................................... 206
    7.3.2 Analysis ............................................................. 213
  7.4 Conclusion .................................................................... 214

8 Self-organisation and Homoeostasis within an Artificial Organism ........... 216
  8.1 Introduction .................................................................. 216
  8.2 Additional Background .................................................. 217
  8.3 Artificial Organism Model ................................................ 218
    8.3.1 Systemic Analysis .................................................... 218
    8.3.2 Data Organisation within the Artificial Organism .......... 222
    8.3.3 Parameters Setup and Behavioural Analysis ................. 223
  8.4 Experiments and Analysis ................................................ 227
    8.4.1 Organism’s Growth and Anomaly Detection ................. 227
    8.4.2 Analysing Self-Organisation Further ......................... 233
    8.4.3 Analysis ............................................................. 241
    8.4.4 Possible Future Work ............................................... 247
  8.5 Summary .................................................................... 247

9 Visualisation .............................................................................. 249
  9.1 Introduction .................................................................. 249
  9.2 Additional Background .................................................. 250
  9.3 Visualising SC Programs ................................................ 253
    9.3.1 Sample Program: A Simple Fire Spreading Model .......... 253
    9.3.2 Graphic Representation of Models ............................... 256
  9.4 First Case Study: A Bistable Gene Network ......................... 262
    9.4.1 Model Presentation .................................................. 262
    9.4.2 Systemic Analysis .................................................... 264
    9.4.3 Model Behaviour ..................................................... 267
    9.4.4 Visualisations ......................................................... 268
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5 Second Case Study: A MAPK Signalling Cascade</td>
<td>271</td>
</tr>
<tr>
<td>9.5.1 Model Presentation</td>
<td>271</td>
</tr>
<tr>
<td>9.5.2 Systemic Analysis</td>
<td>273</td>
</tr>
<tr>
<td>9.5.3 Model Behaviour</td>
<td>275</td>
</tr>
<tr>
<td>9.5.4 Visualisations</td>
<td>278</td>
</tr>
<tr>
<td>9.6 Other Interesting Situations</td>
<td>285</td>
</tr>
<tr>
<td>9.6.1 Systems grouping</td>
<td>285</td>
</tr>
<tr>
<td>9.6.2 Recursive hierarchies</td>
<td>286</td>
</tr>
<tr>
<td>9.7 Analysis</td>
<td>288</td>
</tr>
<tr>
<td>9.8 Conclusion</td>
<td>291</td>
</tr>
<tr>
<td>10 Conclusions</td>
<td>293</td>
</tr>
<tr>
<td>10.1 Summary of Work</td>
<td>293</td>
</tr>
<tr>
<td>10.2 Critical Evaluation</td>
<td>296</td>
</tr>
<tr>
<td>10.3 Objectives Revisited</td>
<td>299</td>
</tr>
<tr>
<td>10.4 Contributions</td>
<td>303</td>
</tr>
<tr>
<td>10.5 Future Work</td>
<td>307</td>
</tr>
<tr>
<td>10.6 Closing Words</td>
<td>309</td>
</tr>
<tr>
<td>A Platform Environment</td>
<td>311</td>
</tr>
<tr>
<td>A.1 Introduction</td>
<td>311</td>
</tr>
<tr>
<td>A.2 SC Compiler</td>
<td>312</td>
</tr>
<tr>
<td>A.3 Command Line Virtual Machine</td>
<td>313</td>
</tr>
<tr>
<td>A.4 Graphical Runtime Environment</td>
<td>315</td>
</tr>
<tr>
<td>A.4.1 Virtual Machine GUI</td>
<td>316</td>
</tr>
<tr>
<td>A.4.2 Visualiser</td>
<td>318</td>
</tr>
<tr>
<td>A.5 Artificial Neural Network Application</td>
<td>328</td>
</tr>
<tr>
<td>B SC Programming Reference</td>
<td>332</td>
</tr>
<tr>
<td>B.1 Introduction</td>
<td>332</td>
</tr>
<tr>
<td>B.2 SC Language</td>
<td>332</td>
</tr>
<tr>
<td>B.3 C++ Platform Library Interface</td>
<td>339</td>
</tr>
<tr>
<td>References</td>
<td>343</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Artificial and natural cellular automata ........................................ 19
1.2 A system used primarily for data storage ...................................... 24
1.3 A system acting as context of interaction ...................................... 24

2.1 A Turing machine reading a tape .................................................. 36
2.2 Shannon’s diagram of a communication system ............................... 38
2.3 Von Neumann architecture ............................................................ 40
2.4 Mono-processor, multi-core and multi-processor processing unit simplified architecture .................................................. 52
2.5 Beowulf computer cluster ............................................................... 53
2.6 Structure of the POEtic tissue ......................................................... 58
2.7 Architecture of a POEtic chip ......................................................... 59
2.8 Pesavento’s automaton implementing von Neumann’s universal constructor .................................................. 80
2.9 Langton’s loops .............................................................................. 81
2.10 Conway’s game of life ................................................................. 83
2.11 Attractors ...................................................................................... 84
2.12 Stochastic $\pi$-calculus model of a HCl molecule ......................... 91
2.13 Model of a chemical reaction using a Petri net ............................... 93
2.14 Model of an exclusive or (X-or) Boolean operator using a Petri net ... 93
2.15 Model of a simple switch using a statechart .................................. 94
2.16 Statecharts notations ................................................................. 95
2.17 Simplified uninterpreted bigraph .................................................. 95
2.18 Bigraph modelling interacting agents in a structured environment .... 96
2.19 Enzyme-substrate complex model using BioAmbients ................... 99
2.20 Graphical representation of a P system ....................................... 100
2.21 Bio-graphs notations and example .............................................. 103

3.1 Properties hierarchy showing the main natural properties and various expressions studied in this thesis .................................................. 111
3.2 Human-program interaction in the context of the systemic computer .... 113
3.3 Graph notations for interactions and corresponding calculus notations .............................................................................. 117
3.4 Graph notations for locality and corresponding calculus notations .... 119
3.5 Molecular model of HCl (Hydrogen Chloride) ................................ 120
3.6 Graphical representation and corresponding notation of the stochastic $\pi$-calculus model of HCl .................................................. 121
3.7 Systemic computation graph model and corresponding notation of the HCl model .................................................. 121
3.8 Structure of a conventional program and of an SC program ............ 122
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9</td>
<td>Systemic computation calculation of print(((a_1 - a_2) \times (a_3 - a_4)))</td>
<td>123</td>
</tr>
<tr>
<td>3.10</td>
<td>A genetic operator acts as a context for two interacting solutions</td>
<td>126</td>
</tr>
<tr>
<td>3.11</td>
<td>Solutions initialisation within an SC model of a genetic algorithm</td>
<td>126</td>
</tr>
<tr>
<td>3.12</td>
<td>Structure of the genetic algorithm model</td>
<td>127</td>
</tr>
<tr>
<td>4.1</td>
<td>System memory</td>
<td>136</td>
</tr>
<tr>
<td>4.2</td>
<td>Graph notation for the sample program</td>
<td>146</td>
</tr>
<tr>
<td>4.3</td>
<td>Graph notation for an interaction between MySystem and MyOtherSystem in the context of MyContext where MyOtherSystem joins the scope of MySystem</td>
<td>153</td>
</tr>
<tr>
<td>4.4</td>
<td>Graphical runtime environment of the SC platform</td>
<td>154</td>
</tr>
<tr>
<td>5.1</td>
<td>Structure of the genetic algorithm model</td>
<td>157</td>
</tr>
<tr>
<td>5.2</td>
<td>Progressive initialisation of a genetic algorithm</td>
<td>159</td>
</tr>
<tr>
<td>5.3</td>
<td>Scheme of a self-adaptive approach for a genetic algorithm on an SC architecture</td>
<td>160</td>
</tr>
<tr>
<td>5.4</td>
<td>An optimal TSP tour through the fifteen largest cities of Germany</td>
<td>162</td>
</tr>
<tr>
<td>5.5</td>
<td>Data organisation within a solution system for a GA solving the TSP</td>
<td>163</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparative test of the two selection methods</td>
<td>168</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparative test of the operators all together and individually</td>
<td>169</td>
</tr>
<tr>
<td>5.8</td>
<td>Operators’ fitness distribution over time for the crossover operators PMX and OX</td>
<td>170</td>
</tr>
<tr>
<td>5.9</td>
<td>Operators’ fitness cubic polynomial fit over time when working all together</td>
<td>171</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of the potentially best tuned methods</td>
<td>172</td>
</tr>
<tr>
<td>5.11</td>
<td>Genetic operators evaluation window’s size over time</td>
<td>173</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparison of the adaptive method with the best tuned methods</td>
<td>173</td>
</tr>
<tr>
<td>5.13</td>
<td>Properties exploitation diagram for a genetic algorithm implemented in SC</td>
<td>176</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of the genetic algorithm model</td>
<td>181</td>
</tr>
<tr>
<td>6.2</td>
<td>Interaction between an external unexpected phenomenon and a program system in the context of the laws of physics and within the scope of the phenomenon</td>
<td>183</td>
</tr>
<tr>
<td>6.3</td>
<td>Two program systems interacting within a context of self-maintenance</td>
<td>184</td>
</tr>
<tr>
<td>6.4</td>
<td>Experiment 1: GA progression without and with faults averaged over 10 runs using a minimalist system configuration</td>
<td>187</td>
</tr>
<tr>
<td>6.5</td>
<td>Experiment 2: GA progression without and with faults averaged over 10 runs with a program configuration using redundant systems</td>
<td>188</td>
</tr>
<tr>
<td>6.6</td>
<td>Experiment 3: GA progression without and with faults averaged over 10 runs with a configuration using redundant systems and facing a strong fault probability</td>
<td>190</td>
</tr>
<tr>
<td>6.7</td>
<td>Experiment 4: GA progression with no fault, with faults and with faults and self-maintenance, all averaged over 10 runs with configurations using redundant systems and facing a strong fault probability</td>
<td>191</td>
</tr>
<tr>
<td>6.8</td>
<td>Properties exploitation diagram for an SC program able to demonstrate fault-tolerance with self-repair</td>
<td>194</td>
</tr>
<tr>
<td>7.1</td>
<td>Axon-dendrites interaction in the context of a synapse within the scope of a particular synaptic connection</td>
<td>198</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>SC model of a neuron showing the dendrites-axon interaction in the context of a membrane and within a soma</td>
</tr>
<tr>
<td>7.3</td>
<td>SC artificial neural network made of 1 neuron and taking 2 inputs</td>
</tr>
<tr>
<td>7.4</td>
<td>SC artificial neural network made with 4 inputs, 1 output and 3 areas sharing neurons</td>
</tr>
<tr>
<td>7.5</td>
<td>SC diagram of a neuron with 3 inputs showing where data to be memorised can be stored in the systems</td>
</tr>
<tr>
<td>7.6</td>
<td>Flowchart of the error backpropagation and delta weight update</td>
</tr>
<tr>
<td>7.7</td>
<td>Error backpropagation example</td>
</tr>
<tr>
<td>7.8</td>
<td>SC feed forward and recursive neural network</td>
</tr>
<tr>
<td>7.9</td>
<td>SCANN application</td>
</tr>
<tr>
<td>7.10</td>
<td>Two networks for the XOR problem</td>
</tr>
<tr>
<td>7.11</td>
<td>Feed forward 4-2-3 network for iris plant data classification</td>
</tr>
<tr>
<td>7.12</td>
<td>A simple perceptron and a perceptron with a recurrent connection from its output to one of its inputs</td>
</tr>
<tr>
<td>7.13</td>
<td>Properties exploitation diagram for an artificial neural network implemented in SC</td>
</tr>
<tr>
<td>8.1</td>
<td>Food to waste cycle for an organism within its environment</td>
</tr>
<tr>
<td>8.2</td>
<td>Food to Waste cycle within the organism</td>
</tr>
<tr>
<td>8.3</td>
<td>Systemic organisation of the organism</td>
</tr>
<tr>
<td>8.4</td>
<td>Organism’s inner organisation over a run of 5000 samples on the UCI breast cancer dataset</td>
</tr>
<tr>
<td>8.5</td>
<td>Ideal dataset’s data repartition</td>
</tr>
<tr>
<td>8.6</td>
<td>Organism’s inner organisation over a run of 3000 samples on the ideal dataset</td>
</tr>
<tr>
<td>8.7</td>
<td>Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C3.3, C4, and C5</td>
</tr>
<tr>
<td>8.8</td>
<td>Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C4</td>
</tr>
<tr>
<td>8.9</td>
<td>Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C5</td>
</tr>
<tr>
<td>8.10</td>
<td>Properties exploitation diagram for an artificial organism implemented in SC</td>
</tr>
<tr>
<td>9.1</td>
<td>Graphical runtime environment of the SC platform with visualisation enabled</td>
</tr>
<tr>
<td>9.2</td>
<td>Structure and behaviour of a simple fire spreading model</td>
</tr>
<tr>
<td>9.3</td>
<td>2D graph views of the fire spreading model at various stages</td>
</tr>
<tr>
<td>9.4</td>
<td>3D explorer views of the fire spreading model in its initial state</td>
</tr>
<tr>
<td>9.5</td>
<td>3D explorer views of the fire spreading model at various stages</td>
</tr>
<tr>
<td>9.6</td>
<td>3D structure views of the fire spreading model at various stages</td>
</tr>
<tr>
<td>9.7</td>
<td>Large scale fire after total combustion using 3D structure views</td>
</tr>
<tr>
<td>9.8</td>
<td>Bistable gene network obtained by an evolutionary procedure</td>
</tr>
<tr>
<td>9.9</td>
<td>Interactions and structure of a bistable gene network model</td>
</tr>
<tr>
<td>9.10</td>
<td>Change in the amount of proteins over time in the two possible states of the bistable gene network</td>
</tr>
<tr>
<td>9.11</td>
<td>Bistable gene network visualisation using the 3D structure with its envelope</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>9.12</td>
<td>MAPK cascade principle</td>
</tr>
<tr>
<td>9.13</td>
<td>Distribution in scope areas of kinase and phosphates</td>
</tr>
<tr>
<td>9.14</td>
<td>Structure of a MAPK cascade model</td>
</tr>
<tr>
<td>9.15</td>
<td>Small MAPK cascade model visualised with the 3D explorer and the 3D structure</td>
</tr>
<tr>
<td>9.16</td>
<td>Evolution in the MAPK cascade of the amount of kinases over time</td>
</tr>
<tr>
<td>9.17</td>
<td>Tree ring view of the three kinases in their activated state</td>
</tr>
<tr>
<td>9.18</td>
<td>Visualisation over time of the MAPK model at three early stages using the 3D structure with its envelope</td>
</tr>
<tr>
<td>9.19</td>
<td>Visualisation over time of the MAPK model at two later stages using the 3D structure with its envelope</td>
</tr>
<tr>
<td>9.20</td>
<td>Visualisation over time of the MAPK model at an advanced and at a late stage using the 3D structure with its envelope</td>
</tr>
<tr>
<td>9.21</td>
<td>Involvement of kinases in phosphorylation after 475 interactions</td>
</tr>
<tr>
<td>9.22</td>
<td>Example of systems grouping within a given scope using the 3D explorer</td>
</tr>
<tr>
<td>9.23</td>
<td>Example of systems captured during execution using the 3D structure</td>
</tr>
<tr>
<td>9.24</td>
<td>2D graph of a model with a recursive hierarchy</td>
</tr>
<tr>
<td>9.25</td>
<td>Example of exploration in depth using the 3D structure</td>
</tr>
<tr>
<td>9.26</td>
<td>Bistable gene network implementation using the stochastic π-calculus</td>
</tr>
<tr>
<td>A.1</td>
<td>SCLab lite mode</td>
</tr>
<tr>
<td>A.2</td>
<td>SCLab full mode</td>
</tr>
<tr>
<td>A.3</td>
<td>SCLab full mode - Visualiser panel only</td>
</tr>
<tr>
<td>A.4</td>
<td>SCANN application</td>
</tr>
</tbody>
</table>
# List of Tables

1.1 Comparative list of computational properties natively found in natural computation and conventional computation ........................................... 22
1.2 Respective contributions from Bentley and from this thesis to the research on systemic computation .............................................................. 28
2.1 Comparative list of computational and behavioural properties found in natural computation and conventional computation ......................... 51
2.2 Evaluation of the natural properties achieved by the reviewed hardware architectures .............................................................................. 61
2.3 Evaluation of the natural properties achieved by the reviewed software approaches .................................................................................. 85
2.4 Evaluation of the natural properties achieved by the reviewed computational paradigms .......................................................................... 108
3.1 Systemic calculus notation ........................................................................................................................................................................... 116
3.2 Investigated models and aim of their analysis .......................................................................................................................................... 129
4.1 Compression code ......................................................................................................................................................................................... 138
4.2 Valid combinations for the wildcard or operator used on labels ........................................................................................................... 139
7.1 XOR experiments results .............................................................................................................................................................................. 209
7.2 Average percentage of success of the iris data sets and total percentage of success .................................................................................... 211
7.3 Average classification value for the Setosa and Versicolour sets with a simple perceptron and an amplified perceptron ............................ 212
8.1 Percentage of the average and standard deviation of data creating a danger signal per class from the breast cancer dataset for various setups .................................................................................................................. 230
8.2 Percentage of the average of data creating a danger signal per class from the breast cancer dataset for the two algorithms presented in [Bentley et al., 2005] and using various setups .................................................................................................................. 231
8.3 Percentage of the average and standard deviation of data creating a danger signal per class from the ideal dataset for various setups .................................................................................................................. 236
8.4 Average percentage of data creating a danger signal and created bridges per class from the wine dataset for various setups .................................................................................................................. 240
9.1 Bistable gene network’s reactions rates ....................................................................................................................................................... 264
9.2 Bistable gene network’s interactions ....................................................................................................................................................... 265
9.3 Bistable gene network’s final state distribution for the SC model and the stochastic π-calculus model ........................................................................ 268
<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.4</td>
<td>Principle of phosphorylation and dephosphorylation in the MAPK cascade</td>
<td>272</td>
</tr>
<tr>
<td>9.5</td>
<td>List of reactions involved in the MAPK cascade</td>
<td>273</td>
</tr>
<tr>
<td>9.6</td>
<td>MAPK cascade interactions</td>
<td>274</td>
</tr>
<tr>
<td>10.1</td>
<td>Evaluation of the behavioural natural properties achieved by systemic computation</td>
<td>305</td>
</tr>
<tr>
<td>10.2</td>
<td>Properties exploitation scheme showing for each behavioural property what computational property was exploited</td>
<td>306</td>
</tr>
</tbody>
</table>
## List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Classical genetic algorithm</td>
<td>68</td>
</tr>
<tr>
<td>2.2</td>
<td>Gradient backpropagation algorithm for feed-forward networks</td>
<td>72</td>
</tr>
<tr>
<td>2.3</td>
<td>Ant simulation algorithm</td>
<td>75</td>
</tr>
<tr>
<td>2.4</td>
<td>Particle swarm optimisation algorithm</td>
<td>76</td>
</tr>
<tr>
<td>2.5</td>
<td>The dendritic cell algorithm</td>
<td>78</td>
</tr>
<tr>
<td>2.6</td>
<td>Cellular automaton algorithm</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>Virtual machine’s engine algorithm</td>
<td>134</td>
</tr>
<tr>
<td>5.1</td>
<td>Pseudo-code of a genetic operator’s function within a GA for the TSP</td>
<td>165</td>
</tr>
<tr>
<td>8.1</td>
<td>Growth context binding method</td>
<td>223</td>
</tr>
</tbody>
</table>
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Definition of a non-context system using a word length of 4 and the first two characters of the kernel as function code</td>
<td>140</td>
</tr>
<tr>
<td>4.2 Definition of a context system using a word length of 4 and the first two characters of the kernel as function code</td>
<td>141</td>
</tr>
<tr>
<td>4.3 An SC program’s encoding section providing a code file, a list of plugins and defining a word length of 4 and the first two characters of the kernel as function code</td>
<td>142</td>
</tr>
<tr>
<td>4.4 Systems instantiations and scopes setup</td>
<td>143</td>
</tr>
<tr>
<td>4.5 Full sample program</td>
<td>144</td>
</tr>
<tr>
<td>4.6 Interaction function declaration</td>
<td>151</td>
</tr>
<tr>
<td>4.7 Interaction function implementation</td>
<td>152</td>
</tr>
<tr>
<td>4.8 Example of a simple interaction function injecting a system into another</td>
<td>152</td>
</tr>
<tr>
<td>A.1 Set of values classified as positive in the X-Or example</td>
<td>331</td>
</tr>
<tr>
<td>A.2 Set of values classified as negative in the X-Or example</td>
<td>331</td>
</tr>
<tr>
<td>B.1 Interaction function prototype</td>
<td>339</td>
</tr>
<tr>
<td>B.2 Inclusion of the SCSystem header file</td>
<td>339</td>
</tr>
<tr>
<td>B.3 Plugin initialisation and exit functions</td>
<td>340</td>
</tr>
<tr>
<td>B.4 Structure of a plugin’s header</td>
<td>340</td>
</tr>
<tr>
<td>B.5 SCSystem public methods</td>
<td>340</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 General Introduction

What is computation? A biological brain can solve complex daily tasks [Kandel et al., 1991], yet does that make it a computer? An organism grows and develops according to its gene pool and its environment [Darnell et al., 1990], is that a kind of program execution? Is the unconscious collaboration of insects with the reproduction of plants and the co-evolution of the species [Whitney and Glover, 2007] an optimised multi-objective algorithm solving two natural problems at once? Are the emerging pigment patterns on the shells of molluses [Meinhardt, 1995] the outcome of a natural cellular automata [Wolfram, 2002], as illustrated in Figure 1.1? Using conventional definitions of computation, the answer to these questions may be no. However, even if the computation of nature is not exactly the same as in our technology, nature clearly seems to be performing some sort of computation.

There are countless examples in nature. Billions of cells communicate and specialise to constitute living organisms [Darnell et al., 1990]. Billions of neurons interact to form the human brain [Kandel et al., 1991]. Proteins, cells, organs, and tissues can work together to elaborate immune systems [Goldsby et al., 2002]. Ant colonies are driven by individual agents acting individually and randomly, yet they can accomplish complex and precise tasks [Dorigo and Stützle, 2004]. Flocks of birds or shoals of fish can split when encountering an obstacle, then move towards a source of food with no individual knowledge or understanding but giving an illusion of a specific group behaviour [Rennard, 2002]. Species evolve and co-evolve all around the world to become fitter and fitter to an ever changing complex environment [Buchanan, 2000, Darwin, 1859, Darwin and Wallace, 1858]). In human society, a crowd in a stadium can create

\[1\] Conus textile photo ©2005 Richard Ling
Chapter 1 Introduction

(a) Cellular automaton rule 30

(b) Seashell *Conus textile*

**Figure 1.1:** Artificial and natural cellular automata

A wave, randomly starting from a couple of individuals greeting a good action, that might unfold or not. Also the build up of new riots might eventually lead at some edge-point to a revolution [Buchanan, 2000]. These *natural systems* are perfect examples of highly complex self-organising systems performing tasks through a massively parallel distribution of sub-tasks, showing homoeostatic behaviour, and yet with no centralised method of control. They present a kind of computation radically different from the conventional computation view based on contemporary computer architectures [Bentley, 2007a] as their behaviour emerges from the massive interaction of sub-elements [Holland, 1998] rather than from a clearly defined sequence of instructions.

The capabilities of nature are such that biological systems have become highly significant to computer science and the past decades have seen the appearance and development of now major fields like evolutionary computation [Holland, 1975] or artificial life [Rennard, 2002].
However, the principles behind the way natural systems work are not only present in organisms; they seem to be ubiquitous in our world [Buchanan, 2000]. The financial market is another example of a complex distributed system in which humans are the agents. The fluctuations are hardly foreseeable and financial crisis show that such system is complex and way beyond our control [Buchanan, 2000]. Solutions to improve the financial market modelling and simulations do not only investigate mathematical solutions but also involve bio-inspired systems [Brabazon and O’Neill, 2006]. In geology, complex crystal structures develop from a nucleus with the addition of new atoms, ions, or polymers [Levi and Kotrla, 1997]. In geophysics, tectonic activity such as earthquakes cannot necessarily be predicted, yet looking back they can be explained [Buchanan, 2000]. A pile of sand at critical state may collapse by the addition of a single grain, yet when slowly adding grains the very instant of the avalanche and its size remain extremely hard if not impossible to predict [Buchanan, 2000]. Finally in science, the structure of scientific revolutions [Kuhn, 1962] is another example which, just as the previous ones, share similar working patterns and can therefore be considered as a natural system.

The quest for the understanding of such systems therefore goes beyond just explaining the principles of biological organisms. As surprising as this might be, the behaviour of an organism or those of the world economy could be more closely related than one would think at a first glance. There are thus strong insights that the understanding of natural systems probably lies within a global answer.

From a technological aspect, a better understanding of natural systems is also of great relevance for computer science. Whilst computers are becoming smaller, cheaper and more interconnected, software still regularly crashes, top of the line robots break down on the wrong kind of ground, power distribution networks fail under unforeseen circumstances [Bentley, 2005b]. With the increasing performance, potential and complexity in machines and software, it has become increasingly difficult to ensure reliability in systems without the addition of significant duplication of modules and redundancy [Avižienis, 1985, Horning et al., 1974]. Yet biological systems demonstrate capabilities such as adaptation, fault-tolerance, self-repair and self-organisation within highly complex systems. For example, in nature, old and potentially damaged cells are constantly being replaced and DNA repaired [Darnell et al., 1990]. The lifespan of cells is shorter than the life of an organism, so fault-tolerance and self-maintenance are essential for the survival of the organism. The failure of some components does not destroy the overall organism; cell death is an important part of staying alive.

From these observations, it is clear that a biologically inspired approach to the handling and management of modern computer technologies may be increasingly desirable. But
how can useful natural features be achieved in computers? The von Neumann architecture [Burks et al., 1946, von Neumann, 1945, 1988], the model of all our conventional computers, implements the well understood concept of Universal Turing Machine [Turing, 1936] using three separate units: the control unit, the arithmetic logic unit and the memory. A computer program stored in memory is executed as an instruction stream by the control unit, relying on the arithmetic logic unit to perform arithmetic and logical operations, and using memory to store data. Such architecture is by essence sequential and relies on a centralised unit of decision, as opposed to systems of natural computation such as the brain, based on a massively parallel architecture with no centralised control unit. Thus even though computers are becoming more and more powerful, the apparent opposition between the two concepts of computation tends to make them incompatible.

Just, for instance, as the development of Prolog [Colmerauer and Roussel, 1996, Roussel, 1975] enabled elegant and precise implementations of logical expressions, so the development of a paradigm where systems could be defined in a manner that resembles their true structures would improve our ability to implement, understand and exploit nature-inspired systems.

To address these issues, [Bentley, 2007a] introduced Systemic Computation (SC), a new model of computation and corresponding computer architecture designed for the modelling and the analysis of natural processes and aiming at enabling a clear formalism of natural systems.

Systemic computation being central to the thesis, the next section provides an overview, then followed by the hypothesis and objectives of the thesis.

1.2 Systemic Computation

The work in this thesis investigates and analyses the properties and capabilities of systemic computation. This section presents the motivation and rules of SC. A review of the early work about SC is given in Chapter 2. Chapter 3 will later present the version of SC used in this thesis as well as notations and methodology.

Systemics is a recent branch of science developed to study systems from a holistic perspective. Following Aristotle’s definition from the “Metaphysics” the term holism summarises the idea that the whole is more than the sum of its parts [Aristotle, 1995].

This vision can be opposed to traditional reductionist approach. Reductionism dates back to ancient Greek philosophy and provides a more mechanical vision of complex systems by considering them as the sum of their fundamental parts which can separately
be studied to additively contribute to the understanding of the whole. In his “Discours de la méthode” (Discourse on the method) where Descartes presents his method to approach a problem, the second precept (out of four) says “to divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution” [Descartes, 1637].

While systemics does not reject the notion of sub elements it emphasizes that interactions between elements are impacted by their context and that elements of a same environment can impact each other. It therefore acknowledges subtle intricate and entwined relationships between individual elements of a whole at various levels of detail and abstraction, making the isolated study of elements insufficient to a global understanding of this whole [Eriksson, 1997, Le Moigne, 1994].

Systemic computation [Bentley, 2007a] is a novel method of computation based on systemics and incorporating at its core natural computation properties as opposed to conventional computation paradigms. Table 1.1 lists native computational properties respectively found in natural and conventional (von Neumann architecture based) computation [Bentley, 2007a, Le Martelot et al., 2007b].

**Table 1.1:** Comparative list of computational properties natively found in natural computation and conventional computation

<table>
<thead>
<tr>
<th>Natural computation</th>
<th>Conventional computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stochastic</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Parallel</td>
<td>Serial</td>
</tr>
<tr>
<td>Continuous</td>
<td>Batch</td>
</tr>
<tr>
<td>Distributed</td>
<td>Centralised</td>
</tr>
<tr>
<td>Approximate</td>
<td>Precise</td>
</tr>
<tr>
<td>Embodied</td>
<td>Isolated</td>
</tr>
<tr>
<td>Circular causality</td>
<td>Linear causality</td>
</tr>
<tr>
<td>Local knowledge</td>
<td>Global knowledge</td>
</tr>
</tbody>
</table>

From this listing the differences and incompatibilities between the two forms of computation become clear. To overcome these issues, models of natural processes that run on conventional computers must simulate the features of natural computation. This often leads to slower and less straightforward implementations compared to analytical or linear algorithms for which computers are well suited [Bentley, 2007a].

Instead of the traditional centralised view of computation, in systemic computation all computation is distributed. There is no separation of data and code, or functionality into memory, control unit and arithmetic logic unit.
Systemic computation stresses the importance of structure and interaction, supplementing traditional reductionist analysis with the recognition that circular causality, embodiment in environments and emergence of hierarchical organisations all play vital roles in natural systems. Systemic computation makes the following assertions [Bentley, 2007a]:

- Everything is a system.
- Systems can be transformed but never destroyed or created from nothing.
- Systems may comprise or share other nested systems.
- Systems interact, and interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems, and systems may have partial membership within the scope of a system.
- Computation is transformation.

The last assertion allows us to call the sorting of pebbles on a beach, the transcription of proteins, or the growth of dendrites in the brain, valid forms of computation. Such a definition is important, for it provides a common language for biology and computer science, enabling both to be understood in terms of computation.

In systemic computation, everything is a system, and computations arise from interactions between systems. Two systems can interact in the context of a third system. All systems can potentially act as contexts to determine the effect of interacting systems. Systems are defined and identifiable by a shape. Context systems make use of these shapes to identify the systems allowable for interaction. The shape of a context system also describes the nature of the interaction it defines [Bentley, 2007a].

In a digital environment, one convenient way to represent and define a system (i.e. its shape) is as a binary string [Bentley, 2007a]. The choice of binary strings for the systems however remains a suggestion of implementation compatible with current electronic hardware. Representing and defining the systems of SC using alternative computational units like with DNA-computing [Adleman, 1994] or reaction-diffusion computing [Adamatzky and De Lacy Costello, 2002] would lead to a different type of implementation.
Each system (i.e. here string) is divided into three parts: two schemata and one kernel [Bentley, 2007a, Le Martelot et al., 2007a]. These three parts can be used to hold anything (data, typing, etc) in binary as shown in Figure 1.2.

![Figure 1.2](image)

Figure 1.2: A system used primarily for data storage. The kernel (in the circle) and the two schemata (at the end of the two arms) hold data.

The primary purpose of the kernel is to define an interaction result (and also optionally to hold data). The two schemata define which subject systems may interact in this context as shown in Figure 1.3. The schemata thus act as shape templates, looking for systems matching their shape and the outcome of the interaction between two interacting systems depends upon the context of this very interaction. A same pair of systems interacting in a different context would lead to a different transformation. How templates and matching is done precisely is explained later in Chapter 4.

![Figure 1.3](image)

Figure 1.3: (a) A system acting as a context. Its kernel defines the result of the interaction while its schemata define allowable interacting systems. (b) An interacting context. The contextual system $S_c$ matches two appropriate systems $S_1$ and $S_2$ with its schemata and specifies the transformation resulting from their interaction as defined in its kernel.

Thus, a system is made of three elements: two schemata that can define the allowable interacting systems in the context of the current system, and a kernel which defines the
nature of the transformation two interacting systems can undergo. Therefore the overall behaviour of a process is emerging from the interactions and transformations of systems in various contexts, rather than from a predetermined sequence of instructions, thus providing a more appropriate and realistic basis for the modelling of natural processes. Also such interaction involves circular causality (systems reciprocally affecting each other at once) as opposed to linear causality (only one system affecting the other at a time) [Wheeler and Clark, 1999]. The aim is no longer to identify long chains of causes and consequences but to take a global and collaborative approach towards the solution to a given problem. In society people can support each other; in nature species can co-evolve like some insects and flowers where flowers provide food for the insects while the insects transport the pollen from flowers to flowers thus allowing their reproduction [Whitney and Glover, 2007]. Whether the systems are people or species, they get transformed by each other and their interaction thus has some effect on both of them. Circular causality is therefore a crucial feature of natural systems, yet currently largely ignored in their modelling, making the models inaccurate.

Finally, systemic computation exploits the concept of scope [Bentley, 2007a]. Whenever two systems interact, they both belong to a certain space, range or scope making this interaction possible. Two people can talk face to face if they are in the same room, not if they are in a different place. The scope of their interaction would here be the room they are both in. The seas of the Earth interact with the Moon because they are in the scope of the Moon’s gravitational field. Considering cellular automata, cells can interact with their neighbours (fixed amount of cells determined by the topology of the cellular automata). In systemic computation systems can potentially interact with all the systems sharing a same scope. Each system can be scope for other systems. The nature of scopes can be modified over time by other systems, or interactions can change the scopes of systems. Therefore the interaction potential per system is defined by the state of the model rather than by a fixed topological limitation, offering a more flexible approach to the notion of scope.

Thus a system can contain or be contained by other systems. Interactions can only occur between systems within the same scope. Therefore any interaction between two systems in the context of a third implies that all three are contained within at least one common super-system, where that super-system may be one of the two interacting systems.

From a more philosophical approach, systems can also be regarded as holons [Koestler, 1989], being a whole but also parts (building blocks) of a bigger whole, being autonomous, self-reliant and independent, affecting and being affected by a hierarchy (hierarchy).
1.3 Hypothesis and Objectives

The hypothesis of this thesis is:

**Systemic computation enables exploitation and analysis of natural properties within nature-inspired computer models.**

In this work, the following definitions are used:

**Systemic computation (SC)** is a new model of computation and corresponding computer architecture, introduced in [Bentley, 2007a], based on a systemics world-view and supplemented by the incorporation of natural characteristics (see Section 1.2).

For the purpose of this work, **natural properties** are properties of natural systems. They can be classified into computational properties and behavioural properties. From a computational view natural computation can be regarded as being stochastic, asynchronous, parallel, continuous, distributed, approximate, embodied, having circular causality, and relying upon local knowledge. From a behavioural view natural systems can be described as being homoeostatic, robust, fault-tolerant, autonomous, and open-ended. This differentiation of properties is useful because a computer model can have the aforementioned computational properties and yet not show for instance homoeostatic behaviour. These properties are reviewed and defined in Chapter 2.

The **exploitation** of natural properties is the use of natural computational properties as building blocks to provide a model with behavioural properties. This exploitation is limited for conventional programs for they are powerless when undergoing an operating system crash, itself powerless when undergoing a hardware failure. Such programs could therefore never be crash-proof. SC provides a design with no distinction between hardware and software, thus not undergoing the same limitations. This thesis focusses on the exploitation of stochastic, asynchronous, continuous and distributed computation relying upon local knowledge and investigates robustness (under the forms of self-adaptation and flexibility), fault-tolerance (under the forms of basic fault-tolerance, crash-proof computing and self-repair), homoeostasis and self-organisation behavioural properties.

The **analysis** of natural properties assesses the use of SC to help design models with natural properties and to understand the natural properties within those models (e.g. how and why an SC model of an artificial organism can show homoeostatic behaviour). This thesis focusses on two types of analysis. First, the systemic analysis aims at analysing the natural properties and organisation of a natural system, and expressing them in SC (described further in Section 3.6). Then, the behavioural analysis aims at analysing the role, the impact, the interaction, the changes in state, the change in hierarchical organisation of the systems, as well as the flow of information. This thesis presents tools that
enable low-level, short-term behavioural analysis and high-level, long-term tendencies to be identified and compared between models.

**Nature-inspired computer models** are computer models intended to encapsulate processes or properties of a natural system through simulation. This work focusses on individual-based models [Grimm and Railsback, 2005], as opposed to mathematical models [Rutherford, 1994] where one or more equations may be used to summarise the behaviour of the natural system as a whole without simulation of the internal workings of the system. This work splits nature-inspired computer models in two categories: nature-inspired algorithms (e.g. genetic algorithms, artificial neural networks, artificial immune systems, artificial ant colonies [Bentley, 2002]) and models of nature (e.g. tumour growth [Wheldon, 1988], models of the liver [Kriván and Wasserman, 2001], gene regulatory networks [de Jong, 2002]). The scope of this work is limited to the modelling of genetic algorithms, artificial neural networks, artificial organism, bistable gene network, and mitogen-activated protein kinase cascade.

This thesis has the following main objectives:

- Assess the work done in the related areas (e.g. natural and unconventional computation, biological modelling) and the work done by Bentley with SC.

- Determine which natural properties (computational and behavioural) can be feasibly achieved (e.g. robustness, fault-tolerance, homoeostasis, self-organisation).

- Formalise the rules, notation, graphical expression of SC and compare with other approaches (e.g. stochastic $\pi$-calculus).

- Identify the models that can best exploit the selected properties.

- Create a complete SC platform (language, compiler and virtual machine) as a proof-of-concept implementation to carry out experiments on modelling and behaviour of programs within SC.

- Perform a systemic analysis and implement those models in SC.

- Analyse the exploitation of the properties and assess the advantages and/or disadvantages of using SC for such exploitation and analysis.

- Investigate visualisation methods for the on-line analysis of SC programs allowing a high-level analysis of information flow and structure within nature-inspired models.
### 1.4 Contributions to Systemic Computation

The work presented in this thesis assesses the validity and utility of systemic computation introduced in [Bentley, 2007a]. It is the first investigation of the exploitation and analysis of natural properties enabled by SC in nature-inspired models. Table 1.2 respectively lists the prior work conducted by Bentley and the contributions from the work presented in this thesis.

<table>
<thead>
<tr>
<th></th>
<th>Bentley</th>
<th>Thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prior work</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original SC concept</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Original SC graphical notation</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Original SC programming language definition</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Original low-level implementation</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Original SC Hardware design</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><strong>Methodology</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Systemic analysis</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SC calculus notation</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Comparison of SC with the stochastic $\pi$-calculus</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>Platform</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full SC programming language definition</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Full SC platform (compiler &amp; virtual machine)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SC runtime environment</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SC visualisation framework</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><strong>Nature-inspired models</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genetic algorithm (GA)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Artificial neural networks (ANN)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Artificial organism</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Bistable gene network (BGN)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Mitogen-activated protein kinase (MAPK) cascade</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><strong>Exploitation and Analysis</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Self-adaptation</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Fault-tolerance</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Basic Fault-tolerance</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Crash-proof computing</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Self-repair</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Homoeostasis</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Self-organisation</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><strong>Visual Analysis</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information flow</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Organisational changes</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
1.5 Thesis Outline

Chapter 2 presents a literature review discussing the place of this work within contemporary nature-inspired paradigms and technologies. The background related to more specific fields involved in specific chapters is discussed in the relevant chapters. Chapter 3 defines the methodology used for this thesis. It involves the investigated properties, the equipment that was developed for this investigation, the investigated models and the methods to analyse and assess them. Chapter 4 presents the first complete high level platform implementing systemic computation, including programming language, compiler and virtual machine. Then, to investigate and analyse natural properties several implementations of nature-inspired algorithms are provided. These models illustrate advanced and desirable features provided by SC but also how SC enables ease, clarity and fidelity in the modelling of nature-inspired systems. In Chapter 5 a genetic algorithm model solving the travelling salesman problem is used to investigate self-adaptation. Then flexibility of model evolution is illustrated showing how a feature like self-adaptation of evolution can be added with the minimum of additional code. In Chapter 6, a genetic algorithm is reused to show how systemic computation enables fault-tolerance, fundamental property of natural computing and highly desirable feature in modern computational systems. The work investigates the achievement of fault-tolerance from the crash-proof computing feature to the integration of self-repair. Chapter 7 presents a model of an artificial neural network model, designed to exploit local knowledge and asynchronous computation, significant natural properties of biological neural networks and naturally handled by SC. Exploiting these built-in properties, which come for free, the model enables neural structure flexibility without reducing performance. Chapter 8 investigates the properties of self-organisation and homeostasis with an original kind of program, an artificial organism, involving an artificial innate immune system. The organism relies on a metabolism that can eat data, self-organise upon these data, expel waste, and shows excellent abilities to detect anomalies in its diet. Chapter 9 introduces systemic computation visualisation as a high level of abstraction graphical mapping of the SC formalism. The visualisation tool thus created aims at enabling high level of abstraction analysis of complex systems. It is tested with two natural systems: a bi-stable gene network and a mitogen-activated kinase cascade, and compared against stochastic \( \pi \)-calculus models with graphical extension. Finally Chapter 10 summarises what has been achieved in this thesis and provides a critical evaluation of the work. To finally conclude it then revisits the objectives and contributions, and discusses future work.
1.6 Publications

The work presented in this thesis has resulted in the refereed publication of five international conference papers, a book chapter and two journal papers, listed below.


• Erwan Le Martelot and Peter J. Bentley. Novel visualisation and analysis of natural and complex systems using systemic computation. Information Visualization, To appear, 2010
Chapter 2

Computational Biomimetism,
the Inspiration from Nature for Computation

*The sciences do not try to explain, they hardly even try to interpret, they mainly make models.*

– John von Neumann

2.1 Introduction

Nature has long been a source of inspiration for computer scientists. From the cradle of computer science, originating in the work of Turing and von Neumann [Turing, 1936, von Neumann, 1945], the potential of biology for computing machines was already discussed. In the late 1940s, von Neumann, architect of contemporary conventional computer [Aspray, 1989, 1990, Burks et al., 1946], dedicated a considerable part of his last years to the idea of automata [Shannon, 1958, von Neumann, 1961].

The automata theory is the science that studies abstract machines and the problems they are able to solve. An automaton is an abstract model of a state machine, a machine with a finite or infinite number of possible states and transitions between these states. Despite its mathematical aspect the theory of automata is defined by Shannon as an interdisciplinary science merging mathematics, with logic and Turing machine theory, engineering, aiming for non-numerical applications, and biology, for instance neurophysiology [Shannon, 1958]. It is also related to quantum theory since the appearance of quantum finite automata [Moore and Crutchfield, 2000]. Applications of automata theory include biological phenomena in machines such as adaptation, self-reproduction and self-repair [Shannon, 1958].
An important part of von Neumann’s research with automata investigated how to build reliable machines out of unreliable components [von Neumann, 1956] and how to build self-reproducing machines [von Neumann, 1966]. The former tackled the problem of creating complex systems with a controlled probability of malfunctioning but built from components with a positive probability of failure. Nature can do it. An organism is a large and stable system made of unreliable components, the cells, replicating, doing potential copy mistakes and dying [Darnell et al., 1990]. The latter investigated the creation of increasing complexity by successive generations of machines constructed by parent machines. Nature did it, as suggested by the theory of evolution of species [Darwin, 1859, Darwin and Wallace, 1858].

Von Neumann was also interested in the relation between computers and the brain [von Neumann, 1958]. According to Shannon he was well aware of the important differences between the two architectures: on the one hand computers made of transistors and vacuum tubes, accurate in computation; on the other hand brains made of neurons, less accurate in computation but orders of magnitude more numerous. These differences according to him lead to significant differences in the organisation of the two systems. He indeed thought that while computers are good at sequential computation, brains presumably operate on a more parallel basis but are designed to confront problems requiring less sequential calculation [Shannon, 1958, von Neumann, 1958].

Considering his late work, it is therefore clear that von Neumann was aware of the limitations of conventional computers and believed that alternative views for computation were also valuable.

This thesis investigates an alternative view of computation, systemic computation, and more specifically its potential regarding natural properties within nature-inspired computer models. Systemic computation is not the only model of computation to emerge from studies of nature and properties of natural computation can be found in various nature-inspired algorithms or paradigms. This chapter explores the literature of the related fields, where nature plays an influential role. This influence impacted science and technology in various areas, the following sections therefore provide a review organised by domain of contribution.

First Section 2.2 provides the theory and definitions of computation necessary for a critical analysis and assessment of the work reviewed and the work presented in this thesis. Then Section 2.3 reviews hardware architectures able to support natural computation better than strictly conventional computers, and bio-inspired hardware especially designed for natural computing. Afterwards, Section 2.4 reviews software techniques designed to provide natural properties, whether for software reliability or problems solving.
Finally Section 2.5 reviews alternative paradigms of computation including the systemic computation work that has been achieved prior to this thesis.

2.2 Theory and Properties of Computation

2.2.1 Introduction

In this thesis, the notion of computation is defined as a generic concept referring to any form of information processing. However information can be processed in various ways. An organism exploiting nutrients from its environment in order to survive does not seem to be concerned with memory accesses and arithmetic calculations, yet both seem to have the potential to process information, whether from a natural or a digital environment. This work is concerned with computation according to the natural sense (e.g. as performed by natural systems in their ecosystem), and opposes it to computation in the conventional sense (e.g. as performed by personal computers). This section will thus review the foundations and define the properties of these two forms of computation, revealing their computational dichotomy and therefore leading to a definition of both forms of computation in terms of computational properties.

2.2.2 Foundations of Conventional Computation

In the history of computation, the first known calculating tool can be dated back to more than 2400 BC with the abacus that was used for arithmetic tasks [Ifrah, 2001] supposedly by the Babylonians. Computation has thus a very long history. This thesis being concerned with modern computing, this review will focus on the modern computing era.

The first form of computer, as opposed to calculator, originates in the work of Babbage with his analytical engine [Ifrah, 2001] first described in 1837. For the first time the concepts of automatic calculation and programmability were brought together. The analytical engine was mechanical general-purpose and thus the first programmable computer.

However, modern computing only appeared in the mid twentieth century with Shannon’s work on digital computation [Shannon, 1940], based upon Boole’s work on the Boolean algebra [Boole, 1848]. The concept of modern computers originates in the notions of Turing machine [Turing, 1936] and von Neumann architecture [von Neumann, 1945]. These notions are central to the definition of modern conventional computers and are now presented in chronological order of appearance.
Turing Machine and Turing Completeness

The concept of conventional computer is grounded in the work of Turing who introduced his Turing machine as an abstract device manipulating symbols of a finite alphabet and mechanically performing instructions described by these symbols [Turing, 1936].

In spite of its impact on modern computing the aim of the Turing machine concept was initially to assess the extent and limitations of mechanical computation, and thus to define computability. A classical problem in computability theory is known as the halting problem [Davis, 1958]. This problem asks whether it is possible, given a program description and a finite input, to know if the program will halt or run forever given that input. Turing proved using his machine that the halting problem cannot be solved by a general algorithm, and is therefore undecidable over Turing machines. The set of functions computable by a Turing machine, called Turing-computable functions, thus excludes at least the set of undecidable problems like the halting problem.

An implementation for such machine is commonly described using a tape, a head, a finite table of instructions and a state register, as shown in Figure 2.1.

1. The tape is divided into cells where each cell contains a symbol from a finite alphabet. This alphabet contains at least a blank symbol (say ’0’) and then as many symbols as required.

2. The head reads a symbol from the tape and can shift the tape of one cell to the left or to the right. Note that the tape is assumed to be infinitely extendable on both sides.

3. The table of instructions tells the machine what to do given the machine state and the symbol being read. The potential actions are:
   - Erase or write a symbol on the tape, then
   - Shift the head left or right or stay on the spot, then
   - Change the machine state as described in the table or stay in the current state.

4. The state register holds the state of the machine. There are a finite amount of possible states.

While such machine is able to process data according to a given set of instructions, it can especially be given a set of instructions and symbols that can simulate the behaviour of any other Turing machine. Such machine is then called universal Turing machine (UTM) and lays down an abstract definition for all conventional computers. A formal theory
of computation establishing the equivalence between the concepts of computability and Turing machine (among others) is commonly known as the Church-Turing thesis [Kleene, 1952].

The concept of universal Turing machine led to another fundamental concept in computability theory known as Turing completeness. A computational system is said to be Turing complete if it is able to simulate any UTM, or in other words if it is able to compute every Turing-computable function. A Turing complete system is thus universal with respect to Turing-computable functions.

Therefore, a computational system, in order to demonstrate its universality with respect to Turing-computable functions, and its equivalence in terms of computational power to other Turing complete systems, has to be shown Turing complete, thus equivalent at a certain level of abstraction to a universal Turing machine. Such demonstration can be achieved by showing how a computing system can simulate one specific UTM, which as a consequence implies that the system can simulate any UTM, and is thus Turing complete. Cellular automata rule 110 was demonstrated to be Turing complete in this way by showing how the cyclic tag system, known to be Turing complete, could be emulated using this rule [Cook, 2004].

Whilst the Turing paradigm (notions of Turing machine and Turing completeness) provides crucial concepts for computer science, it relies on assumptions that can be subject to debates. Such machines are logical, yet a physical implementation is required and as such has consequences. The concept of universality provides one machine and methods to fit all problems but a dedicated implementation to match a given problem can also be beneficial (e.g. faster computation, easier implementation). Conventional systems have a pre-determined state space but computation processes can potentially expand or restrain such space. A system could also potentially be in several superposed states rather than in a single state [Stepney et al., 2005].
Shannon and the Foundations of Digital Computing

A crucial breakthrough for electronics was presented in Shannon’s 1937 master’s thesis. He initially demonstrated that Boolean algebra and binary arithmetic could be used to better arrange relays in telephone routing switches. He then reversed the idea and proved that relays can be arranged to solve Boolean algebra problems [Shannon, 1940]. Using this concept, electrical switches could thus be used to perform logic operations. So was born digital electronics, followed by digital computers.

A few years later Shannon published another groundbreaking contribution [Shannon, 1948] that laid down the foundations of information theory. In this work information is treated like any quantity, and can thus potentially be manipulated by a machine. This paper introduced the basic elements of communication by providing measures for information quantity, its compression limits and its robustness when sent over a noisy channel. By combining this view of information with his earlier work with Boolean logic for digital circuit he developed a model of communication based on a binary system. The work introduced the term *bit*, standing for *binary digit*, which due to the Boolean nature of digital computing is the basic unit of information storage and communication.

To measure the amount of information contained in a message, or the rate at which information is produced, Shannon introduced a definition of entropy as given by Equation 2.1:

$$H(X) = -\sum_{x \in X} p(x) \log_b p(x). \quad (2.1)$$

where $X$ is a discrete random variable representing messages, $X$ the set of all messages $x$ that $X$ can be, $p(x)$ their respective probability and $b$ the base of the logarithm used (the base for compression into bits being 2).

Given this measure, Shannon provided the source coding theorem that establishes the limits of information compression by giving the optimal average length of the encoded message. The theorem states that the average coded message’s length $L$ for the random variable $X$ is given as in Equation 2.2:

$$H(X) \leq L(X) < H(X) + \frac{1}{N} \quad (2.2)$$

where $N$ is the amount of symbols in the source alphabet (e.g. 26 letters of the English alphabet plus spaces and punctuation symbols).

With these tools, information could be encoded in an optimised way provided a noiseless transmission. However, physical transmission, whether across wires or radio frequencies, is subject to noise. To model a general communication system Shannon provided the diagram given in Figure 2.2. This diagram involves:
Figure 2.2: Shannon’s diagram of a communication system

- An information source producing the message;
- A transmitter turning the message into a signal;
- A channel carrying the signal and subject to noise that can affect the signal;
- A receiver turning the signal back into a message;
- A destination receiving the message.

The aim of information transmission must therefore address the problem of reproducing at one end of a noisy communication channel a message formulated at the other end. To efficiently transmit a message across such system, the message encoding method thus has to take into account the potential signal corruption. Shannon introduced the noisy-channel coding theorem, describing the potential of error-correcting methods provided the channel capacity. The channel capacity is the maximum amount of information the channel can reliably transmit (rate of reliable transmission). The theorem is formulated as follows:

*Let a discrete channel have the capacity $C$ and a discrete source the entropy per second $H$. If $H \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with an arbitrarily small frequency of errors (or an arbitrarily small equivocation). If $H > C$ it is possible to encode the source so that the equivocation is less than $H - C + \epsilon$ where $\epsilon$ is arbitrarily small. There is no method of encoding which gives an equivocation less than $H - C$.*

Claude Shannon [Shannon, 1948], Theorem 11, p22
While this theorem does not provide a coding method, the notion of redundancy is introduced stating that “It is clear [...] that by sending the information in a redundant form the probability of errors can be reduced.” Following the noisy-channel coding theorem, this is however only valid provided $H \leq C$ as otherwise an arbitrarily small probability of error is not achievable.

Shannon’s communication theory was born out of his research for the transmission of information. The theory addressed efficient source message coding as well as robustness within unreliable transmission environments, particularly for the transmission of information over a noisy channel in a sequential manner. Whilst providing measures for information it does not define information itself, making the theory applicable to any form of information. Using this approach information could be reduced to a basic form made of a sequence of bits and thus processed by a machine implementing Boolean algebra. As a result this theory became another crucial part of the foundation of modern computation.

**Von Neumann Architecture**

The first computers called program-controlled computers appeared in the forties. The Colossus [Copeland, 2006] was first shown to be working in 1943. It was the first programmable digital electronic binary language based computing machine, although not Turing complete. Then in 1946, the ENIAC (Electronic Numerical Integrator And Computer) [Goldstine, 1972], the first general purpose Turing-complete electronic computer was announced. It could be programmed to perform most of what modern algorithms use: sequences, loops, arithmetic operations, inputs and outputs and conditional branches. However, these machines were programmed by changing switches, re-wiring and re-building parts which made the process very arduous and could take weeks [Copeland, 2006].

Research was being conducted to develop computers that would be simpler to use. A considerable breakthrough in the architecture of computers was introduced in [von Neumann, 1945] with the EDVAC (Electronic Discrete Variable Automatic Computer). This computer was working with a binary system and would keep its program and data in a storage space. Its architecture became known as the von Neumann architecture, or the stored-program computer. It was a major advance over previous architectures as such computer no longer required physical reconfiguration in order to execute various programs. It also implements a universal Turing machine.

Such architecture is organised in four parts: the arithmetic logic unit (ALU), the memory, the inputs and outputs (I/O) and the control unit. It is illustrated in Figure 2.3.
Figure 2.3: Von Neumann architecture

- **Arithmetic Logic Unit (ALU):** the ALU performs arithmetic operations, logic operations and comparison operations [Mano, 1992, Stallings, 2005, von Neumann, 1945]. Arithmetic operations include addition, subtraction, and sign swapping but might also include multiplication, subtraction or modulo, trigonometric functions such as sine or cosine, and square root functions. Bit-shifting operations can also be performed but are equivalent to multiplication and division by two. Logic operations are the Boolean logic operations such as the AND, the OR, the XOR and the NOT. Comparison operators include equality, superiority or inferiority tests between two numerical values.

- **Memory:** Memory can be described as a sequence of cells where each cell contains a small quantity of information, a single number. This information can be used to indicate to the computer what to do, these are the instructions, or to hold data to process. Each cell is addressable by a number (or address) and can be read (e.g. instruction or data) or written on (e.g. write data) [Mano, 1992, Stallings, 2005].

- **Inputs and Outputs (I/O):** The I/O devices, also called peripherals [Mano, 1992], enable communication between the computer and the outside world [Eadie, 1968]. Such devices include the mouse, the keyboard and the screen but hard drives disks, optical drives disk (e.g. CD, DVD players), graphics cards, sound cards and computer networks are also forms of I/O. These I/O devices can be complex and assist the CU is particular tasks. Graphics cards for instance include their own memory and dedicated processor to compute fast 3D graphics [Shirley, 2002].

- **Control Unit (CU):** The CU reads instructions from the memory at the location indicated by the instruction pointer [Stallings, 2005]. The instruction pointer is initially placed at the beginning of the executed program. The instructions describe what operations should be submitted to the ALU and potentially what
to do upon the result. Once the current instruction executed, the CU moves on to the next instruction indicated by the instruction pointer. The next instruction is by default the instruction following the previous one in the memory, unless the previous instruction contained another order (e.g. branch, jump, subroutine call, return) requiring to move and read from elsewhere in the memory. The CU is therefore the communication manager between the ALU, the memory and the peripherals, and is in charge of sequencing operations, making such architecture by essence serial (or sequential).

Modern conventional computers (e.g. personal computers), although more powerful and complicated (e.g. multi-core, involving complex devices) are still based upon this architecture. They are commonly organised in central processing unit (CPU), memory and peripherals [Mano, 1992]. The CPU groups ALU, CU, small fast memory called registers and an I/O unit. The memory part mainly includes the random access memory (RAM) allowing stored data to be accessed in any order, as opposed to other mediums like optical discs which reading is closely related to the momentum of the head. All these parts are linked by computer buses [Stallings, 2005], parallel cables grouped together to transmit several bits at a time between devices.

Considering the functioning of such machine the mechanical and mathematical origins of this technology are clearly reflected. The mechanical aspect inherited from the Turing machine [Turing, 1936] is found in the sequential processing of instructions, where a machine, here the processor, receives a sequence of inputs, the instructions, to produce a sequence of outputs, like calculation results written to memory. The mathematical aspect lies in the basic operations, performed on numbers, here binary [Shannon, 1948, von Neumann, 1945], and on the precise order imposed on the operations to guarantee a valid result. If considering the four numeric values $a$, $b$, $c$, $d$ and the two instructions $a := b + c$, then $b := a + d$ where $:=$ is the affection operator, the order of execution matters for the resulting values of $a$ and $b$. The simultaneous execution of both instructions has an undefined result. Therefore such architecture imposes a sequential computation.

2.2.3 Properties of Conventional Computation

Computational Properties

In this work, conventional computers are defined as the computers based on a von Neumann architecture (the architecture of most modern computers) such as personal computers and implementing a universal Turing machine. As a result, conventional computation is the computation as performed by conventional computers: sequences of
instructions underlying arithmetic and Boolean logic implemented at a low-level with logic gates composing large digital circuits.

To analyse the heritage of these mechanisms on computation, computation can be defined in terms of properties. As discussed in this chapter, in [Stepney et al., 2005] and [Bentley, 2007a] the Turing and von Neumann paradigms are imposing limitations on computation due to the assumptions they make. Table 1.1 (page 22) lists computational (i.e. native) properties of conventional computation as suggested in [Bentley, 2007a, Le Martelot et al., 2007b]. Conventional computation is/has:

- *Deterministic* as conventional programs are not relying on randomness;
- *Synchronous* as the execution of instructions is timed by a processor clock;
- *Serial* as the instructions are executed one at a time by a processor;
- *Batch* as programs are designed to terminate (as opposed to natural processes such as evolution);
- *Centralised* as programs are executed by a central unit with a centralised memory;
- *Precise* as its foundations lie in mathematics which provide rigour and infinite precision (as opposed to the natural world which has no concept of mathematics and where systems vary quantities such as chemicals, cells or individuals);
- *Isolated* as computers are not structurally coupled to their environments, constantly affecting and being affected by them (as opposed to embodied systems [Quick et al., 1999] such as living organisms that can be significantly altered in morphology by their environments, like plants while growing or connectivity in the brain, and that can alter their environments);
- *Linear causality* as instructions work upon the concept of implication, where $a$ affects $b$ (as opposed to systems where $a$ and $b$ affect each other simultaneously).
- *Global knowledge* as any specific instruction can potentially access any part of the memory and depend on global system information.

**Behavioural Properties**

While behavioural properties relate to the nature of a program and thus its code, the nature of programs itself is strongly impacted by the aforementioned computational properties. Conventional computation can be stated as being natively:
- **Externally organised** as computer systems rely in order to function upon the explicit design of their programmers and these systems cannot change or adapt their structure without being explicitly programmed for;

- **Heterostatic** as computers are not able to maintain their functionality under unforeseen circumstances (i.e. the failure of a part makes the machine out of order);

- **Brittle** as the compartmentalisation in units and tasks centralisation within von Neumann architectures makes the machine unlikely to function properly when a unit or a communication between units is damaged (as opposed to decentralised systems such as living organisms);

- **Fault intolerant** as programs can crash or stop working properly without any possibility of graceful degradation when encountering a single unexpected configuration (e.g. segmentation fault, division by zero, hardware failure);

- **Human-reliant** as computers are designed to be programmed by humans (as opposed to natural processes which do not require human intervention to function properly);

- **Limited** as conventional computer programs do not grow in complexity over time (as opposed to the evolution of organisms from monocellular to multicellular organisms with differentiated cells);

- **Simple** as the outcome of a program is clear and predictable (as opposed to complex systems with emergent effects preventing the outcome from being predictable).

The term *natively* is used here to define properties that are direct consequences of the underlying computational properties, thus without any dedicated software engineering involved in a program or specific hardware to alter or provide a given property.

As should be clear, the behavioural properties of conventional computers provide no native abilities to robustness and autonomy. With programs getting very large and thus more subject to programming mistakes, and considering the difficulty to always predict all the potential inputs (e.g. a robot exploring a space), this can yield significant issues in the software or machine behaviour. For some of these issues, software engineering solutions can be provided to address, fully or partially, the issue. This will be addressed in Section 2.4. However some issues require more than software engineering and alternative hardware only can then provide solutions. Such solutions are reviewed in Section 2.3.

Conventional computation and all its aforementioned properties contrast significantly with natural computation (see Table 1.1, page 22) which will now be analysed from foundations to properties.
2.2.4 Foundations of Natural Computation

To disambiguate the use of the term *natural computation* it is defined in this work as the nature of the computation performed within natural systems, whether biological or physical, and should not be confused with the field of research this thesis fits in.

Natural systems are another example of computing systems. They process some inputs, like nutrients, oxygen or light, transform them, for instance into energy or proteins, and potentially expel waste like carbon dioxide. Yet they process a different form of information that has no concept of mathematics, Boolean logic and binary digits. Chemical bonds between atoms and molecules depend upon their respective chemical properties, their exchange or sharing of electrons, leading to specific compound shapes like lattice or snowflake [Levi and Kotrla, 1997, Pauling, 1960]. These chemical bonds thus dictate the structure of chemical compounds, which can in turn be processed and transformed into other compounds by other systems. In genetics, genes are sequences of DNA bases that can synthesise specific proteins which can in turn regulate the synthesis of specific genes they can bind to. This binding process involves three-dimensional structural elements called structural motifs located on the proteins and defining DNA-binding domains allowing to recognise specific DNA sequences [Lilley, 1995]. Computation is thus here performed by transformation of compounds, defined by their chemical shape. The shape (e.g. proteins, genes) defines their role (e.g. what proteins binds to what gene) and therefore the structure defines the functionality.

For the purpose of this work, *information* within natural systems (e.g. molecule, protein, gene) is considered as being equivalent to *structure* and *functionality*. This definition enables natural systems to be seen as information processors, and as such, computers.

However, the functioning of natural systems is clearly different. Instead of relying upon clearly defined sequences of instructions, natural systems rely on numbers of *interactions* of sub-elements [Holland, 1998] (e.g. attractive forces between atoms and molecules, genes regulation by proteins) enabling the *emergence of complexity* (e.g. complex global behaviour emerging from simple local rules) at various levels of organisation. As a result of these working principles the *architecture of natural systems* is significantly different from the architecture of conventional computers. These four central notions are now defined.

**Interactions**

In chemistry, interactions of chemical substances compose and decompose chemical compounds, changing their structure and properties. These interactions involve the motion
of electrons which creates and breaks chemical bonds [Pauling, 1960]. In physics, elementary particles interact by exerting forces on each other through the exchange of particles carrying the fundamental forces of nature [Gribbin, 2000]. In biology, as previously discussed specific proteins can bind to specific genes and their interaction can yield the synthesis of other proteins [Lilley, 1995].

In this work, an interaction between two or more systems (e.g. objects, elements) is an event where the interacting systems affect one another. The result of an interaction between systems is a transformation of both these systems at the same time. This notion therefore differs significantly from causal principles stating that a given system affects another system, which might in turn affect the first one [Wheeler and Clark, 1999].

Considering two pool balls hitting each other, they mutually and simultaneously affect each other. The shock between the two balls is the interaction and the resulting transformation is the change in their velocity and position the shock induces.

Also, interactions can be regarded as contextual. The interaction between systems is defined by those systems but also by the context they interact within.

For instance, the interaction of oxygen and hydrogen within water would have a different outcome depending on the pressure and temperature they undergo, resulting in a solid, liquid or gaseous state.

**Complexity and Complex Systems**

The notion of complexity can accept various definition depending on the field, whether philosophy, sociology, physics, biology or computer science, and even within the fields. There is no generic and generally accepted fixed definition of complexity [Edmonds, 1995]. As this work is concerned with natural systems, regarded here as complex systems, complexity will thus be defined here in accordance with complex systems.

In this work, a system is considered to be complex when it is composed of many interacting entities leading to emerging properties not fully-predictable (e.g. computable by an equation modelling the whole system), thus non obvious, from the properties of the constituting elements. A complex system is not necessarily complicated as its structure can be easy to understand. In order to model a complex system, sub-systems and rules operating on them have to be defined in order to run simulations. A non-complex system could on the contrary be described by a set of equations describing its behaviour as a whole (i.e. not just describing the low-level interactions).

A clock mechanism is not complex as its result can be described over time by equations and no simulation or complete experiment is required in order to know the outcome. The brain, gene regulation, the immune system, ant colonies, the evolution of species,
the financial market are complex as it is very difficult to provide a precise prediction about their evolution over time. Such systems are classified as complex adaptive systems which Holland defines as:

*A Complex Adaptive System (CAS) is a dynamic network of many agents (which may represent cells, species, individuals, firms, nations) acting in parallel, constantly acting and reacting to what the other agents are doing. The control of a CAS tends to be highly dispersed and decentralized. If there is to be any coherent behavior in the system, it has to arise from competition and cooperation among the agents themselves. The overall behavior of the system is the result of a huge number of decisions made every moment by many individual agents.*

John Holland [Waldrop, 1992]

Chaos systems [Lorenz, 1963, Poincaré, 1890] are another form of complex systems that defines the set of dynamic systems that are deterministic and recurrent but yet very hard to predict due to their extreme sensitivity to initial conditions. Due to the sensitivity and recurrent aspect of these systems, perturbations grow exponentially and thus make the systems seemingly random.

Therefore complex systems are not predictable. They are also expected to undergo non-linear processes [Heylighen, 2008] such as extreme sensitivity to initial conditions, the famous *butterfly effect* coined from the talk “Predictability: Does the flap of a butterfly’s wings in Brazil set off a tornado in Texas?” [Lorenz, 1972] whose name illustrates the concept. However they display properties, hence some form of order, and are thus not random or chaotic. Complexity is commonly accepted as being located in between order and disorder [Waldrop, 1992]. It is also suggested that complex systems have a spontaneous tendency to evolve towards the edge of order and chaos [Waldrop, 1992]. These observations lead to suggestions that this subtle equilibrium might be the requirement for complex phenomena such as life to occur [Lewin, 2000, Waldrop, 1992].

To obtain such behaviour the entities composing complex systems must be distinct and independent enough, not to make the whole fully predictable, but must also have a certain mutual dependence, without which disorder would reign [Heylighen, 2008]. This form of mutual dependence is defined by the interactions between constituting elements of a system. No interaction would indeed mean a system made of purely independent elements which can only behave randomly on their own.
Emergence

Even though the concept of emergence can be traced back to Aristotle [Aristotle, 1995], the term was first coined by Lewes [Lewes, 1875]. It is closely related to the notions of complexity and evolution in complex or natural systems [Holland, 1998, Kauffman, 1993].

Emergence can be defined as the appearance of new features at a given level of complexity. This is a process by which complex patterns arise from the intertwining of simple rules through multiple interactions. New features defining a higher-level behaviour or organisation from lower-level interactions are emerging properties of the system (e.g. crowd behaviour, cars traffic). As a result, a characteristic of emergence is that the whole is more than the sum of its parts. This implies as second characteristic that the behaviour of the whole cannot be wholly explained by the knowledge of its separate parts.

However, the definition of emergence is still the subject of debates and two forms of emergence can be argued: strong emergence and weak emergence [Bedau, 1997, Chalmers, 2006, Corning, 2002, Koestler, 1969].

Strong emergence [Chalmers, 2006, Laughlin, 2005] defines properties irreducible to the system’s components. This is the most common notion of emergence in philosophical discussions about emergence [Chalmers, 2006]. The appearance of life out of inanimate elements, or the emergence of consciousness from nervous systems, can be seen as strongly emergent phenomena.

Weak emergence [Bedau, 1997, Chalmers, 2006] considers that the dynamics of the whole is wholly determined by the causal dynamics of the parts and that emergence is a subjective interpretation of the external observer regarding appearing and potentially unexpected features. This interpretation is the one commonly used in scientific discussion and work about emergence [Chalmers, 2006]. The work presented in this thesis considers emergence as defined in its weak form.

Architecture of Natural Systems and Self-organisation

Whether considering chemical bonds between atoms and molecules [Pauling, 1960], genes regulated by proteins [Darnell et al., 1990], or ants communicating through pheromones [Dorigo and Stützle, 2004], natural processes are made of collective interacting systems. Yet, these systems are not all from the same level of details. Indeed organisms do interact with each other and their environment, but then a deeper level could consider cells interactions within these organisms, then an even deeper level could consider genes and proteins interactions within the cells, then atoms, then molecules, and so on down
to the quark level. Interactions can therefore be analysed at various successive levels of abstraction.

Each of these abstract levels presents new emerging properties (e.g. the cell level underlies the organ level which in turn underlies the organism level). As defined by the weak emergence [Bedau, 1997], the notion of level remains subjective but is based on concrete observed properties and structures (e.g. organs have emerging properties that cells do not have, weak emergence would thus imply an organ level above the cell level). These interlocked levels thus define embedded hierarchies (e.g. organisms are made of organs themselves made of cells) where each level of detail is an emerging expression of its sub-level (e.g. interactions of organisms result from interactions at the cell level resulting from interactions at the molecular level, etc). These levels, representing phenomena emerging from lower level phenomena, are commonly called integrative levels or levels of organisation and were already discussed in philosophy in the 1920s [Conger, 1925].

The concept of self-organisation within collective systems is thus an outcome of such process in which global structure spontaneously emerges from local interactions [Camazine et al., 2001, Heylighen, 2008]. The spontaneous aspect implies no global control of the process and the outcome is therefore the result of a collective behaviour where no element in particular is indispensable, allowing the death or the replacement of agents without damaging the whole. The nature of the process therefore provides self-organisation and robustness to perturbation and damage as emerging properties of a parallel and distributed system (i.e. collective).

Ant colonies are an example of such organisation where individuals build a civilisation not sensitive to the death of some of them and able to adapt to a changing environment [Dorigo and Stützle, 2004]. In organisms, the constant death and renewal of cells [Darnell et al., 1990] is an important part of the life process. The brain, made of billions of interacting neurons in the case of humans [Kandel et al., 1991], is yet another example of emergence at various levels involving the self-organisation of neurons as a network, leading to more abstract features such as memory, decision making, or consciousness.

2.2.5 Properties of Natural Computation

Computational Properties

Considering the foundations of conventional and natural computations, it is clear that they both operate in a different way, as highlighted in Table 1.1 (page 22) that lists
computational (i.e. native) properties of both conventional and natural computations. Natural computation is/has:

- **Stochastic** as interactions between constituting elements happen randomly and the outcome of the interaction also includes randomness (e.g. two instances of evolution of a natural system from a given initial state would be different and unpredictable);

- **Asynchronous** as interactions are not synchronised by a clock;

- **Parallel** as interactions happen simultaneously between the various elements of the system;

- **Continuous** as natural systems perpetuate tasks to maintain a stable level of organisation (e.g. ants maintaining the colony, evolution; as opposed to “run then terminate” programs normally used in computers);

- **Distributed** as computation is spread over several elements (e.g. neural networks, ant colonies; as opposed to centralised in one or few elements);

- **Approximate** as there is no concept of exact number (unlike in mathematics), quantity and elements can be unstable (e.g. quantum mechanics [Bellissard, 1985]);

- **Embodied** as natural systems are constantly affecting and being affected by their environment (e.g. a plant growing in response to the light levels and nutrients in its environment and affecting the growth of its companions, the human brain wiring itself in response to interactions with its environments and the ability of humans to modify the same environment);

- **Circular causality** as interacting elements affect each other at the same time (e.g. two pool balls hitting each other);

- **Local knowledge** as elements only interact at their level of organisation within a scope of interaction (e.g. neighbourhood for a cell) with no concept of the whole (e.g. a cell is not aware of the organism it constitutes).

**Behavioural Properties**

Based on the aforementioned computational properties the notions of emergence and complexity lead to more advanced features. Natural computation can be stated as being natively:

- **Self-organised** as natural systems form their own architecture without any need of external instruction (e.g. neural structure): this process emerges from multiple
interactions of sub-components relying on local knowledge and with no reference to the global pattern [Camazine et al., 2001];

• **Homoeostatic** as natural systems have feedback mechanisms responding to their inner state in order to maintain themselves in a stable condition (e.g. regulation of internal body temperature, destruction of invading pathogens);

• **Robust** as natural systems can cope with and adapt to varying conditions (e.g. ant colonies can find their way round a new obstacle to restore a path, neurons can rewire);

• **Fault-tolerant** as the failure of elements do not imply the terminal failure of the whole (e.g. cells dying do not prevent the organ from working, ants dying do not affect the global behaviour of the colony) and natural systems can regenerate and replace broken parts (e.g. tissues can grow back) or create new elements (e.g. newly born ants);

• **Autonomous** as natural systems follow their own aim and are thus independent from any external decision (as opposed to computers being human-reliant);

• **Open-ended** as natural processes can grow in complexity over time (e.g. evolution), and thus have no restriction regarding their potential;

• **Complex** as natural systems are built upon the emergence of properties through successive levels of organisation starting at the base from simple rules between simple interacting elements.

### 2.2.6 Summary

Conventional computation and natural computation present two opposite views of computation. These two views are summarised in Table 2.1. Contrary to conventional computation, natural computation presents features such as self-organisation and fault-tolerance leading to more complex features such as robustness. Considering the brittleness of conventional computers, natural computation has appealing features and the motivation for integrating them within modern technologies is clear. Yet, natural computation displays such behaviour inherently (i.e. without being explicitly designed for) and these features are resulting from the computational properties of natural systems, thus emerging from concepts built at their core.

The following section presents hardware approaches to natural computation (i.e. how to get natural properties built into an architecture) and the section after presents software approaches to natural computation (i.e. how to get natural properties within conventional software). The last section reviews alternative paradigms of computation.
Table 2.1: Comparative list of computational and behavioural properties found in natural computation and conventional computation

<table>
<thead>
<tr>
<th>Property Type</th>
<th>Conventional computation</th>
<th>Natural computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational</td>
<td>Deterministic</td>
<td>Stochastic</td>
</tr>
<tr>
<td></td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>Continuous</td>
</tr>
<tr>
<td></td>
<td>Centralised</td>
<td>Distributed</td>
</tr>
<tr>
<td></td>
<td>Precise</td>
<td>Approximate</td>
</tr>
<tr>
<td></td>
<td>Isolated</td>
<td>Embodied</td>
</tr>
<tr>
<td></td>
<td>Linear causality</td>
<td>Circular causality</td>
</tr>
<tr>
<td></td>
<td>Global knowledge</td>
<td>Local knowledge</td>
</tr>
<tr>
<td>Behavioural</td>
<td>Explicitly organised</td>
<td>Self-organised</td>
</tr>
<tr>
<td></td>
<td>Heterostatic</td>
<td>Homoeostatic</td>
</tr>
<tr>
<td></td>
<td>Brittle</td>
<td>Robust</td>
</tr>
<tr>
<td></td>
<td>Fault intolerant</td>
<td>Fault-tolerant</td>
</tr>
<tr>
<td></td>
<td>Human-reliant</td>
<td>Autonomous</td>
</tr>
<tr>
<td></td>
<td>Limited</td>
<td>Open-ended</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>Complex</td>
</tr>
</tbody>
</table>

2.3 Hardware Approaches to Natural Computation

2.3.1 Introduction

While the von Neumann architecture has proven to be an important concept for computer science, it is nevertheless limited and significantly incompatible with natural properties. In that respect, significant research is underway to develop new computer architectures offering different computational properties, whether extending the conventional architecture or providing alternative architectures.

2.3.2 Conventional Hardware Architectures

Multi-core and multi-processor systems provide desktop or laptop computers with several processors instead of one. This enables enhanced performance for multi-applications running. Two computation demanding applications running on a dual core/processor architecture can potentially use one processor each and therefore not be slowed down by the other. Considering the operating system, such architecture thus also provides a better reactivity of the system when undergoing computationally demanding tasks. Figure 2.4 illustrates the architecture of such architectures. While more CPUs provide more computing power the underlying architecture remains unchanged. A memory failure can still cause terminal failure to the operating system. Therefore whether using
one, two or more processors, it remains a traditional computer, faster, but with the same limitations.

Supercomputers [Murray, 1997], introduced by Cray in the 1960s and made of many processors connected by a local high-speed computer bus, are designed to deal with highly calculation-intensive tasks such as physical simulations, molecular modelling or chess playing. They process computation as fast as the technology of the moment can allow. Considering the increasing hardware computation speed, the supercomputers of today are the personal computers of tomorrow. Modern supercomputer architectures can be slit into three categories: multi-processing computer, SIMD (Single Instruction Multiple Data) processors able to simultaneously execute an instruction on several data sets, and computer clusters. The two formers remain super conventional computer and as such face the same limitations. The latter however works differently.

Computer clusters [Castagnera et al., 1994] present a significantly different architecture. While using conventional computers this method links computers, or nodes, together through a network in order to function as a highly powerful and parallel computer. As a result the cluster is no longer a machine run by one central unit and one operating system. It has been used to implement nature-inspired modelling paradigms such as membrane computing systems [Ciobanu and Wenyuan, 2004]. In this context the crash of one machine does not necessarily imply the failure of the cluster. Several types of clusters can be classified, depending on their application, as presented below.
High-availability clusters [Marcus and Stern, 2000] exploit the concept of redundancy to provide a better resistance to crash. In this cluster, each computer has at least one redundant version (i.e. another computer) in the network so that if one fails to provide a service another node can still provide the service. The failure of a node is potentially detected by a heartbeat mechanism which consists in a reciprocal communication between servers aiming at monitoring the status of each other. This technique can therefore show a certain tolerance to faults within limit of the redundant nodes. It might however fail if the heartbeat connections go down, in which case each machine wrongly thinks the others failed and thus duplicates the services which can then cause data corruption within the shared resources.

Clusters have also been designed for demanding applications such as weather or space computational simulations. In this case fast communication is often necessary between the processing units and a dedicated, dense and homogeneous network is thus required to link the computers. Such cluster architecture is called Beowulf cluster [Sterling et al., 1995], as illustrated in Figure 2.5 1. The system usually consists of a server node and several client nodes connected via a network, each running a Unix-like operating system. The server node centralises the control and is the interface between the cluster and the outworld. The client nodes are configured and controlled by the server node. They only perform the tasks they are required to do. Beowulf nodes can be seen as additional CPU and memory units that can be added to the cluster, similarly to CPU and memory units that can be plugged into a motherboard. As a result Beowulf clusters behave like single machines with many processors [Radajewski and Eadline, 1998].

Load-balancing clusters [Bourke, 2001] share a similar working principle. They define a network of machines that can function virtually as one. Each computer on the cluster is

1Public image from the Wikimedia project
running a load balancer software program. In some cases, some machines are servers and others are nodes, or all machines can be servers, known as a server farm. The aim is to increase the performances of server-based services by distributing requests throughout the cluster (e.g., web service requests in case of a web server application). This technique can be used for high-availability clusters.

Finally, grid computing [Joseph and Fellenstein, 2004] is a cluster architecture that relies on a massively parallel computation while assuming little inter-node communication. It is commonly used for solving single problems that can be subdivided in parallel autonomous tasks. The certain independence between nodes enables a more flexible structure as computation relies less on communication and thus nodes need not be located next to each other but can be scattered over and linked by the internet. This flexibility permits volunteer computing [Sarmenta, 1998] which enables any computer owner to join a grid network in order to share their computing resources. Resources can also be rented out by a provider and the computational power, whether processing or memory, is metered like the electricity or water supply. This process is commonly known as utility computing. A similar recent approach is known as cloud computing. Its differentiation from grid computing is still debated but virtualisation of software is argued as being the main difference [Vaquero et al., 2009] as cloud computing imposes lesser constraints than grid computing on the client software (e.g., grid-architecture constraints).

Computer clusters are thus an implementation of powerful, distributed and parallel computation able to use redundancy mechanisms to cope with faults, or potentially survive the loss of a node. However, even though each machine has the potential to be independent, computation tends to remain centralised by one or more servers which can lead to similar brittleness of the whole. A bug in the server may lead to a wrong distribution and management of the tasks, a crash of the server may terminate the application. In addition the diversity and brittleness of operating systems and protocols [Bentley, 2005a] involved in a computer network can make the platform difficult to manage and program for. Despite their computational power, these forms of computers therefore remain anchored in the conventional computer paradigm and do not suit perfectly a massively distributed and decentralised approach such as natural computation.

Peer-to-peer (P2P) [Khan and Wierzbicki, 2008] is another computer network, often used for file sharing, where there is potentially (when implemented purely) no concept of clients or server when all machines on the network are peer nodes playing both the roles of client and server. P2P networks thus have the potential to be fully decentralised. The advantage of such network is to grow its capacity as peers join the network, as opposed
to client-server networks where the addition of clients can slow down the service. The decentralised architecture also implies that the failure of a node does not have to fail the whole network, making it tolerant to faults.

Another type of parallel processing found in most modern computers is the graphics processing unit (GPU). GPUs are highly parallel units that were designed to mainly perform 3D graphics rendering operations (e.g. texture mapping, polygon rendering, matrix transformations) for which they are more efficient than general-purpose CPUs. Offloading such heavy tasks to the GPU also allows the CPU to more efficiently perform the remaining tasks (e.g. operating system’s other tasks, game or simulation engine). GPUs being very powerful and highly parallel, they can also be efficiently used to perform general purpose computation (GPGPU) on tasks where CPUs would perform less efficiently. However their parallelism is limited to performing the same program on all processors with a different data set for each. Yet, nature-inspired algorithms such as genetic algorithms which perform fitness evaluation on all solutions do present such form of parallelisable computation. In that respect several genetic programming implementations have been performed on GPUs, mainly for fitness evaluation [Poli et al., 2008]. [Harding and Banzhaf, 2007] showed that the speed increases for the evaluation of individuals are hundred of times faster than on a typical CPU implementation. [Langdon and Banzhaf, 2008] also presented a GP tree interpreter for GPU and showed that significant acceleration can be gained. Their method enabled a very fast evaluation of a large population on a non-trivial problem (a quarter of a million individual programs in four seconds).

Distributed and parallel computing can thus offer significantly more computational power than monoprocessor computers for applications that can be split into parallel tasks. Highly parallel systems such as natural systems can thus be efficiently simulated on such architectures [Harding and Banzhaf, 2007, Langdon and Banzhaf, 2008, Melab et al., 2006, Poli et al., 2008] or used to improve scalability, adaptability and fault-tolerance on grid-computing [Champrasert et al., 2005]. However the diversity of hardware, software and protocols involved makes the whole still subject to potential failures (e.g. programming mistakes, bugs). Reliability and efficiency could therefore be improved by an integration of the hardware, software and protocols, designed to work as one, rather than reusing and combining material designed for other purposes.

2.3.3 Nature-inspired Hardware Approaches

As the observation of natural systems suggests, von Neumann machines are not the only way to perform computation and electronic chips are not necessarily the only medium
that can potentially be used.

*Ubiquitous computing (or ubicomp)* [Weiser, 1991] (also commonly referred to as *pervasive computing* [Hansmann et al., 2003] and *everyware* [Greenfield, 2006]) presents a different view of computation. It suggests a new human-computer interaction approach by integrating information and computation ability into everyday objects while making it invisible to the user. *Ambient intelligence* [Aarts et al., 2001], built upon ubiquitous computing, defines an electronic environment that can sense the presence of people and be responsive to it. Applications like the *internet of things* [Meloan, 2003] aims at assisting people with their everyday tasks in a natural way. In contrast to a desktop paradigm where a user initiates the interaction with the machine for a well defined purpose, this paradigm therefore relies on *natural interaction* where ambient devices compute for us with no explicit request through a graphical user interface. Computation is here autonomous and omnipresent. The temperature of a house for instance could be automatically regulated room by room by using biometric data measured with devices present in the clothes of all individuals present in the house. Ubiquitous computing therefore defines computation as the massively parallel and decentralised interaction of computing devices continuously exchanging information. Such architecture thus shares by design many computational properties of natural systems (e.g. parallel, distributed, continuous, embodied, or using local knowledge). The world is on the way towards more and more ubiquitous devices assisting us in the everyday life (e.g. mobile phones, global position system) but the technology to fully implement the aforementioned ideas is still the subject of research [Gasson and Warwick, 2007].

In this respect technological advances such as *speckled computing* [Arvind and Wong, 2004] may link the material and the digital worlds with thousands of sprayed on tiny devices. This future technology aims at creating minute, autonomous semi-conductor specks that are programmable, incorporate sensors and communicate through wireless networking. Such technology is intended to support ubiquitous computing. In speckled computing, data is sensed, processed and information is extracted collaboratively in situ [Arvind and Wong, 2004], offering a data-centric paradigm as opposed to the client-server paradigm that could be inadequate for ubiquitous computing [Gaber, 2007].

*Wireless sensor networks* [Culler et al., 2004, Haenselmann, 2006] also offer a way to scatter devices around us. These networks consist of spatially distributed autonomous devices equipped with sensors in order to monitor environmental conditions such as temperature or sound in various locations [Haenselmann, 2006, Römer and Mattern, 2004]. Such technology has been used for bird and wild animals observation, cattle herding, bathymetry, ocean water and grape monitoring, cold chain management, rescue of avalanche victims, vital sign and power monitoring, parts assembly, tracking military
vehicles, self-healing mine field, sniper localisation [Römer and Mattern, 2004]. Sensor networks could also be used for wearable computing [Mann, 1997], devices carried around or worn within clothes, with potential application to the monitoring of elderly people [Haenselmann, 2006]. Such technology therefore also offers a good medium for ubiquitous computing.

Field-programmable gate array (FPGA) is another technology introduced by Xilinx which implements the reconfigurable computing paradigm, aiming at combining the flexibility of software with the high performance of hardware. They are chips with thousands of parallel simple logic processing units whose functionality and connectivity can be reconfigured on-line in microseconds. This is done by writing a configuration program into a static memory and the configuration is then spread across the chip to update it and thus create a functionally new device. This technology has been used in [Jackson and Tyrrell, 2002] to create fault-tolerant hardware, or in the Embryonics project [Tempesti et al., 2007] which investigated self-replicating hardware for reliability. A subclass of membrane computing systems has been implemented in FPGA [Petreska and Teuscher, 2004]. Genetic programming can also benefit from FPGA to speed up fitness evaluation but also to define specialised operators and even for a full implementation [Poli et al., 2008]. In the field of neural networks, a digital neuron model suitable for evolving and growing heterogeneous spiking neural networks on FPGAs was presented in [Shayani et al., 2008].

Using conventional hardware, [Thompson, 1997] investigated evolution within an evolvable program, held in random access memory, designed to control an autonomous mobile robot. His work showed that evolution can produce systems fault-tolerant by nature, especially arguing that insensitivity to mutations gives insensitivity to faults. It also argues that evolution can produce fault-tolerance arising to some degree for free due to the nature of the process, but can also have it explicitly stressed in the fitness function or can even potentially build working parts out of faulty parts.

The field of evolvable hardware [Greenwood and Tyrrell, 2006] also brings together reconfigurable hardware such as FPGAs and evolutionary computation to create fault-tolerant, autonomous and dynamically adaptive systems. This techniques uses evolutionary algorithms to evolve circuits that can be physically tested within reconfigurable hardware (e.g. FPGA) (or simulated in software) until a circuit configuration exhibits the expected behaviour. Such hardware can also potentially change its architecture and behaviour on-line, making it suitable for adaptive systems (e.g. that change on-line depending on their interactions with their environment).
Specific hardware has been especially designed to integrate natural features. The *POEtic* project [Tempesti et al., 2002] aims at creating a platform organised with a similar hierarchy as found in biological systems. It is capable of implementing in digital hardware systems inspired by the three major axes of bio-inspiration identified by the authors: *phylogenesis*, *ontogenesis*, and *epigenesis*. Phylogenesis defines the evolutionary development and history of a species. Ontogenesis refers to the development of an organism according to its genetic code. Epigenesis considers the development of an organism within and influenced by its environment. In this respect the POEtic tissue [Tempesti et al., 2002, Tyrrell et al., 2003] was developed as a self-contained, flexible, and physical substrate able to interact with its environment, develop and adapt through the processes of evolution, growth and learning. Its structure, illustrated in Figure 2.6, reflects the three axes. The phylogenesis layer deals with the evolution of the genetic material, the ontogenesis layer deals with the growth of the tissue from the genetic material and the cellular differentiation, the epigenesis layer deals with the behaviour and development of the organism as it interacts with its environment (e.g. development of the nervous or immune system) [Tyrrell et al., 2003]. The tissue is designed to be flexible and users can implement the layers they need. Such platform is designed to incorporate at its core natural features such as fault-tolerance and self-repair [Tyrrell et al., 2003]. An electronic architecture of a POEtic chip is presented in [Thoma et al., 2004] and illustrated in Figure 2.7. The environment subsystem deals with environment interactions and as such involves sensors and actuators. The organic subsystem deals with the behavioural process of the tissue. The system interface enables a connection between these two parts. The architecture is scalable and therefore single chips and tissues are managed in

---

2 Reproduced with permission from [Tempesti et al., 2002].
3 Reproduced with permission from [Thoma et al., 2004]
the same way. In this implementation the organic subsystem is composed of \textit{molecules}, each of them connected to their four neighbours, and of a cellular routing layer built on top of the molecular structure. The routing layer implements a dynamic routing algorithm fully distributed that allows the creation of path between cells. However the environmental subsystem within a chip is composed of a conventional processor aiming at configuring the molecules through a system interface that communicates with the organic subsystem. As a result the implementation is fully distributed at the organic subsystem level but the management of interactions with the environment as well as the management of molecules remains centralised. This technology has been designed to present features found in living organisms and thus presents natural properties such as distributed and parallel computation able to enable fault-tolerance [Barker et al., 2007] and self-repair (at the molecule level). The hardware implementation allows fast execution speed and a large-scale spiking neural network model has been successfully implemented on it [Moreno et al., 2006].

Another similar project known as \textit{Perplexus} introduced the design of the \textit{ubichip}, a reconfigurable electronic device capable of implementing bio-inspired hardware systems featuring growth, learning, and evolution [Upegui et al., 2007]. It has also been used for neural networks to implement a neural circuit capable of growing by physically creating and destroying synaptic connections based upon neural activity [Upegui et al., 2008].

Other mediums of computation have also been explored and can provide new ways to design massively parallel computing devices from unconventional materials [Adamatzky, 2001].

In the field of chemical computing [Kuhnert et al., 1989] showed how image processing such as contrast modification, contours discerning or smoothing of partially degraded pictures can be performed using light-sensitive chemical waves. Contrary to conventional

\begin{figure}
\centering
\includegraphics[width=\textwidth]{POEtic_chip.png}
\caption{Architecture of a POEtic chip}
\end{figure}
sequential computers, such process can be considered as operating on a parallel basis. Reaction-diffusion computers [Adamatzky et al., 2005] are massively parallel computing devices where concentration of chemicals represent data, and computation is performed by chemical reactions. It has been demonstrated that logic gates could be modelled [Adamatzky and De Lacy Costello, 2002] within reaction-diffusion computing.

Another general approach, the collision-based computing [Adamatzky, 2002], where computation results from the collision of particles, has been used to model and analyse processes from physics, logic or cellular automata. Research on this technology is ongoing and presents potential for new chemical computers in the future.

Evolution in materio of logic gates was performed within liquid crystal [Harding and Miller, 2007] therefore showing that it is possible to program such unconventional material for conventional computation. More complicated tasks such as tone discrimination [Harding and Miller, 2004] and robot control [Harding and Miller, 2005] were also successfully evolved in this medium.

Biochemistry and molecular biology have also been exploited within DNA Computing [Adleman, 1994] using molecules of DNA to encode problems. It exploits the many different molecules of DNA to try many different possibilities at once. It has been applied to solve NP-complete problems such as the directed Hamiltonian path problem. Another forms of molecular computing is known as peptide computing [Hug and Schuler, 2001, Sakthi Balan et al., 2002] and relies on the affinity of antibodies towards peptide sequences in order to perform computation.

Quantum computing [DiVincenzo, 2000] exploits quantum mechanical phenomena, such as superposition and entanglement, to perform parallel computation. Quantum computers maintain sequences of qubits (quantum bits) where each qubit can hold a one or a zero, like classical bits, but also any quantum superposition of these states. As a consequence pairs of qubits can be in any quantum superposition of four states, triplet of qubits in any superposition of eight states, etc. Therefore $n$ qubits can be in any quantum superposition of $2^n$ states, as opposed to conventional bits being in only one of them at once. Quantum computers have quantum logic gates to manipulate the qubits and quantum algorithms to perform and solve problems.

### 2.3.4 Summary

Table 2.2 provides for the reviewed architectures an evaluation of the level of achievement of the natural computational properties, as well as an evaluation of the behavioural properties that the architectures are designed to achieve.
Table 2.2: Evaluation of the natural properties achieved by the reviewed hardware architectures. The evaluation ranges from no dot, if the property is absent, to three dots if the property is fully supported.

<table>
<thead>
<tr>
<th>Property</th>
<th>Conventional hardware</th>
<th>Nature-inspired hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>•</td>
<td>•••••</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Continuous</td>
<td>•••</td>
<td>•••••</td>
</tr>
<tr>
<td>Modular</td>
<td>•••</td>
<td>••••</td>
</tr>
<tr>
<td>Open-ended</td>
<td>••</td>
<td>••••</td>
</tr>
<tr>
<td>Complex</td>
<td>•••</td>
<td>••••••</td>
</tr>
<tr>
<td>Stochastic</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Continuous</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Modular</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Open-ended</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Circular causality</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Local knowledge</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Self-organized</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Homeostatic</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Robust</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Autonomous</td>
<td>•••••</td>
<td>••••</td>
</tr>
<tr>
<td>Open-ended</td>
<td>•••••</td>
<td>••••</td>
</tr>
</tbody>
</table>

Trends in computational paradigms and engineering clearly seem to evolve towards smaller, interconnected and scattered devices. Internet can be predicted as being ubiquitous in the future [Gershenfeld, 1999]. Yet, while computation is increasingly becoming more parallel, decentralised and distributed, its complexity seems to approach the limits of human capabilities [Kephart and Chess, 2003] and such trends come at the cost of control and reliability. In conventional distributed computing, most of the work is adopting a traditional client-server interaction paradigm. However, for ubiquitous and pervasive computing applications this paradigm could be inadequate to involve context awareness, auto-adaptive and emergence functionalities [Gaber, 2007]. For instance, sensor networks are an example of distributed computing using a data-centric paradigm. In such case the sensor providing the data is irrelevant as only the data matters. A request for a temperature measure in an area is no longer a request to a specific device with an address but to an autonomous network able to process the request. The development of alternative computers therefore also suggests the development of novel methodologies reflecting a more natural computation. The next section presents software approaches to natural computation and the section after reviews alternative paradigms of computation.
2.4 Software Approaches to Natural Computation

2.4.1 Introduction

Features of natural systems can be very useful for modern applications. Software reliability can greatly benefit from fault-tolerance ability, search techniques can benefit from distributed self-organisation or self-adaptation to explore large unknown search spaces, pattern recognition can benefit from self-organisation applied to knowledge representation and anomaly detection.

While the behavioural properties within natural systems are a consequence of their computational properties, some natural features can potentially be created within conventional software. Creating artificially such properties supposes that a part of the software is in charge of their management. This can be attempted in a conventional fashion (i.e. without simulating computational properties of nature) or in a natural fashion (i.e. by simulating computational properties of nature).

This section will first review conventional software approaches to natural behavioural properties and will then review alternative ways to obtain such properties with nature-inspired software approaches.

2.4.2 Conventional Software Engineering Approaches

Common Approaches to Robustness and Fault-tolerance Handling in Conventional Software

While human-designed software can execute tasks efficiently and precisely, it usually relies on an ideal physical environment. However, memories are subject to faults, processors can overheat or communications between machines can fail. Operating systems become very unstable if the memory is corrupted. Computers turn themselves off if the processor overheats. The environment of computers is thus unfortunately not ideal and as a result software is brittle [Bentley, 2005b].

In addition software development itself can be a source of brittleness. While correctness proofs can potentially be provided for an algorithm [Gabbay, 1992], programming mistakes can yield unexpected values that might not be properly handled by the software. An access to a forbidden memory area might terminate a program without a proper closure procedure. A badly handled loop might lock the program in an infinite process that has to be killed with sometimes no possible recovery of the work in progress (e.g. the
Software reliability can therefore be difficult to guarantee. Fault-tolerance and robustness are however properties found in natural systems, whether based on self-adaptation to changes to develop alternative solutions or self-repair to rebuild broken parts. To address this issue, several software techniques or architecture solutions have been investigated and developed to help programmers with writing more reliable code.

*Exception handling* [Goodenough, 1975] is a technique commonly used in modern programming languages. Its aim is to handle the occurrence of exceptions in order to alter the regular flow of execution. Exceptions are not qualified as errors as they are expected, thus detected, special situations for which a solution is usually provided. Depending on the language they can be handled using different methods, and different syntax.

In object-oriented programming such as C++, Java or C#, the *try-catch* blocks [Bloch, 2001] allow to perform some computation upon detection of an exception (an error detected and advertised by firing an exception object). It assumes that exceptions are created by a test designed by the programmer to check the validity of the state of the program, or by the potential virtual machine (e.g. stack overflow, out of bounds access in an array). It then has to be handled by alternatives blocks of code. Exceptions can also be *checked* [Bloch, 2001] at compilation by specifying in the methods signature the exceptions they are allowed to raise.

Another way to handle exceptions uses the concept of *design by contract* [Meyer, 2000], like in the Eiffel language. The idea is to place statements at the beginning and at the end of functions, providing pre and post conditions that must be asserted for the program to work properly. Contracts can also be specified for a class (also called class invariant) to assert that some given conditions are always valid. This technique helps to assert that instructions are executed in the right context (e.g. non-empty array, object initialised) to reduce the risk of errors occurring in the method while originating elsewhere earlier in the program. When a contract is violated, some *rescue* code can be executed, similarly to a try-catch block.

Exception handling is a common tool for programmers to get some control over exceptional cases which do not fit within the normal expected flow of operations. A watcher, whether a try block or a contract, tries to catch exceptions intentionally thrown by the user or potentially automatically by the virtual machine (e.g. Java, C#). It is therefore potentially allowing the designer to provide some code aiming at restoring the program in a stable state, and as such relies on redundancy. Depending on where the exception occurred and how it has been anticipated by the designer, the program might be able
to recover or not and potentially crash. Also, in languages such as C++ that have no virtual machine, there is no automatic error check for potential low-level errors like wrong memory access or division by zero. As a result such error would still cause terminal failure to the program without exception raising. Exception handling relies then on testing potential mistakes and throwing exceptions instead of returning some integer error values like in C. It is therefore in such case just an advanced and elegant form of defensive code.

Recovery blocks [Horning et al., 1974], introduced in 1974, aims at providing alternative blocks of code for some blocks of code that might fail to work properly. It relies first upon acceptance tests designed by programmers to evaluate the correctness of the execution of some code, second upon alternative solutions if prior code failed to pass the acceptance test. A recovery block consists of one primary block of code, an acceptance test and zero or several alternate blocks. At the end of the primary block, the acceptance test is performed to assess the code. If the primary block is detected by the test to be failing, then an alternative block is executed and tested using the same acceptance test. If the alternate block is detected as being failing again, another alternate block is executed. Should all alternate blocks fail, then the recovery block has failed. In this case, if this recovery block is a sub-part of a higher recovery block, whether primary or alternate block, the same process applies to the higher level recovery block. This method is therefore based on code blocks redundancy. It requires localising and structuring the code blocks that might fail. It also requires designing efficient assertion tests for each recovery block in order to assess them properly and guarantee their success or failure. It finally requires providing alternative blocks of code, in case of failure of the regular code block, to perform the same function as the failing block but in a different manner to avoid replicating the potential error. Therefore, similarly to exception handling it requires an understanding of what can potentially go wrong in the code. More importantly it requires significant software development overhead. Finally, the acceptance test itself can be prone to mistakes in which case the whole recovery block might be considered as having failed in spite of a correct output.

N-version programming (NVP) [Avižienis, 1985], or multi-version programming, is a software engineering process that was introduced in 1977 to incorporate fault-tolerance. Two or more functionally equivalent programs are generated by different development teams from the same specifications. The underlying motivation is that the independence of the development of each solution reduces the probability of identical software mistakes. These versions are then put together and all executed. Their result is then compared by the NVP framework which is in charge of making the final decision. This decision can be made using the most frequent output or any other criteria. Just as recovery blocks, this method introduces functional redundancy in order to cope with potential error. However
instead of working at the code block level, this method uses redundancy at a higher level, the software level. Similarly to the previous method with its acceptance test, the output is subject to a decision algorithm. In spite of the effort to develop various solutions, there is no guarantee that similar mistakes between teams cannot happen. Furthermore, the decision algorithm that decides of the final outcome from the results of the various executions can make the wrong decision. Therefore, just as the previous method, the significant development cost overhead does not guarantee a correct output.

These methods rely on the anticipation of the potential errors, the detection of these errors and some code redundancy or alternative code to secure the execution or a graceful degradation or a recovery. However it does not guarantee that the alternative programs are not facing the same issues. It can also make mistakes when, in case of errors detected, deciding which program version is providing the right answer. Finally, the development and cost overheads can be important (several programs for one specification). Therefore these techniques, provided significant development overhead, can enable better software reliability. This however only involves a certain tolerance to predictable faults, but not for unexpected situations. There is also no self-repair ability but only, and if explicitly designed for it, a form of recovery ability by allowing alternative code to try another way to perform an operation that failed. Yet, natural systems provide an example of systems able to cope naturally with damage and recover from it.

**Autonomy in Expert Systems and Multi-agent Systems**

An *expert system* [Giarratano and Riley, 2004] is a tool attempting at performing an analysis on a given problem as a human expert would do. It involves a knowledge base, a set of rules, and an inference engine able to derive further knowledge from the acquired knowledge and the rules. The aim is to provide an answer to a given problem using this inference mechanism. It commonly relies on *logic* to build new knowledge: *if P is true* and if the engine has the rule *P implies Q* then a new fact is *Q is true*. The inference rules can be used in two ways. By forward chaining, new knowledge is discovered from known knowledge. By backward chaining, goals are assessed by attempting to work backward and find knowledge allowing to conclude the goals. Also in such system, an answer is not binary but provided with a *confidence factor*, or *certainty factor*, which provides a measure of the confidence in the answer (e.g. an estimated percentage to be right). More rules or knowledge can be added to the expert systems through experience (e.g. asking question to an end user). An expert system therefore works with knowledge and rules that can grow over time to allow a better knowledge. This is made possible as the knowledge they build of a problem is made out of data, as opposed to a program solving a problem using a specific and fixed algorithm.
As such systems are built upon logic, they provide consistent answers and ask all necessary questions to produce an answer. However, this form of knowledge is rigid (e.g. in logic, two similar facts $A$ and $A'$ are distinct premises even though a human might interpret them as very close) and systems can therefore lack common sense. Also they cannot come up with creative advice as they can only derive knowledge. There is thus no deep understanding of the problem [Giarratano and Riley, 2004].

Expert systems have been designed to replace human expertise and has thus been used to perform autonomous tasks in systems like satellite scheduling and control [Barry and Sary, 1989] or electric utility management automation [Fereidunian et al., 2009].

Multi-agent systems (MAS) is another technique that can be defined as systems including multiple autonomous entities with either diverging information or diverging interests, or both [Shoham and Leyton-Brown, 2008]. As such they can be suited to distributed architectures like the internet. The potential role of agents in a MAS spans from trading agents to game playing agents [Shoham and Leyton-Brown, 2008]. Due to their distributed and autonomous properties, MAS also have the potential to show self-organisation [Di Marzo Serugendo et al., 2005].

The peer-to-peer (P2P) architecture previously discussed has been shown to be of particular relevance to MAS and vice versa [Koubarakis, 2003]. P2P has for instance been used to develop a multi-agent system for collaborative management of bibliographic databases [Karoui et al., 2004]. The aim of this work is to assist users in managing their local bibliographical databases and to exchange data among relevant users in an implicit and intelligent manner. Each personal agent can fill in missing data or check data correctness as well as provide recommendations.

MAS is also strongly connected to nature-inspired fields such as ant colony optimisation [Dorigo and Stützle, 2004] or swarm intelligence [Beni and Wang, 1989] that are discussed further in this chapter. The next part will now review software approaches to natural computation by simulation of natural computational properties and nature-inspired software design.

### 2.4.3 Nature-inspired Software Approaches

As previously discussed, natural systems demonstrate behaviours that can be useful for modern applications. However their behaviour is emerging from natural computational properties. As opposed to conventional software approaches the reviewed methods are based on a simulation of one or several computational properties of nature in order to get the desired behavioural property.
Evolutionary Algorithms: Distributed, Parallel, Local, and Autonomous Computation for Problem Optimisation

Since the origins of life, species have evolved to develop from simple unicellular microorganisms to complex multicellular organisms [Bowler, 2003]. Over millennia, generations of individuals reproduced and died, and species developed or went extinct. Each individual in a species is a little different, and the observation of these subtle differences and their consequences on a long term led Darwin and Wallace in their respective research to the theory of evolution by means of natural selection [Darwin, 1859, Darwin and Wallace, 1858]. This theory has been since then unified with genetics as the modern evolutionary synthesis [Kutschera and Niklas, 2004].

The key idea is that organisms self-replicate with slight variations. Individuals with advantageous traits have an improved capacity for survival and reproduction, making individuals in the next generation more likely to inherit these traits, and progressively causing unfit traits to disappear. This process, called natural selection, leads to gradual, slow and favourable changes over time in the populations. The consequences are mainly a regulation of the species population and a constant adaptation of the species to their environment.

Initially, this theory was not linked to genetics as genetics was not known and genes had not been discovered. Natural selection and genetics got combined in the 1930s [Kutschera and Niklas, 2004]. Evolution can thus also be defined as the successive changes in the genetic material of a population of organisms over generations.

Consequently, whether satisficing species feature on the planet scale, or randomly altering the occurrence of alleles in the micro-scale of genetics, evolution seems to present some computational features. Its principles inspired many computer scientists and their work resulted in the development of various evolutionary algorithms: genetic algorithms [Holland, 1975], evolutionary programming [Fogel et al., 1966] and evolution strategies [Rechenberg, 1973, Schwefel, 1981]; the most broadly used being genetic algorithms.

A genetic algorithm (GA) [Holland, 1975] is a global search heuristic method grounded in the aforementioned principles. Whilst inspired by evolutionary biology GAs are using simplified analogies of techniques such as inheritance, mutation, selection, and crossover. GAs commonly ignore development and measures individual fitness directly from its genotype, there is thus little distinction between genotype and phenotype. A given problem is abstracted as a - usually large - search space (all possible genetic combinations) representing all the potential solutions to the problem. Populations of individuals (chromosomes) representing potential solutions (i.e. locations within the search space), are evolved (using crossover and/or gene mutations) throughout successive generations.
using a fitness measure to favour reproduction of the fittest individuals. A typical GA algorithm is given in Algorithm 2.1. Chapter 5 will present a different kind of GA using

**Algorithm 2.1 Classical genetic algorithm**

Randomly generate an initial population
Evaluate the fitness of each individual in the population
repeat
  Select the fittest individuals for reproduction
  Generate a new generation using genetic operations (e.g. crossover, mutation)
  Evaluate the fitness of each individual in the population
until end condition (e.g. fittest individual good enough, maximum time or number of generations reached)

SC and based on local interaction only, without population sorting or creation of new generations at once. GAs are global search methods as the recombination of alleles at reproduction allows to move from a potentially local optima, as opposed to a traditional hill climber method. Although GAs are simple to implement the understanding of their behaviour is not as straightforward [Mitchell, 1996]. Their theoretical foundations have been the subject of much research [Banzhaf and Reeves, 1999, Belew and Vose, 1997, De Jong et al., 2003, Goldberg, 1989, Holland, 1975, Spears and N. Martin, 2001, Stephens et al., 2007, Whitley, 1993, Whitley and Vose, 1995, Wright et al., 2005] and is traditionally explained in terms of building blocks formalised using the concept of schema [Holland, 1975]. GAs have been used in a broad range of applications such as evolving programs (genetic programming [Koza, 1992, Langdon and Poli, 2002]) or cellular automata, performing data analysis and prediction of dynamical systems or protein structure, evolving artificial neural networks’ weights, architecture, or learning rules [Mitchell, 1996].

Evolutionary algorithms rely upon randomness and selection to evolve potential solutions to a given problem until a satisfactory solution is found. The natural computational properties of stochasticity, distribution, local-knowledge, and autonomy lead here to emergent fit solutions to given problems from random solutions, provided a selection criteria. These techniques are also generic as they can apply to any problem as long as encoded according to the algorithm’s requirements. This universality highlights the self-adaptation ability of these techniques to problems search space. This adaptability feature and the distributed aspect of these methods make evolutionary algorithms well suited for exploring large search spaces. There is however no guarantee that they provide a global optima. Dedicated languages and frameworks such as Push and PushGP [Spector, 2001] were also designed and developed to assist implementation of evolutionary algorithms. The application span of evolutionary algorithms is broad and ranges from stock market forecast and optimisation problems to ecosystems simulations [Guan and Mo, 2006, Mitchell, 1996, Rennard, 2002].
Evolutionary Techniques Applied to Robustness in Software design

Natural systems clearly demonstrate another form of design where damage and unexpected circumstances are naturally handled. The functioning of an ant colony is not altered by the loss of some individuals and new individuals will replace them [Dorigo and Stützle, 2004], an organism can survive and recover from substantial damage [Darnell et al., 1990], a plant might even get healthier after being trimmed. Fault-tolerance and self-repair abilities seem to be built-in at the core of the design without any explicit redundant code or fault detection system.

In nature, *development* [Wolpert et al., 2006] is the process by which organisms grow and develop. Natural development uses genes to produce gene-specific proteins, which in turn can regulate the activity of some genes. Performed within a computer the result is an emergent computer program called a *gene regulatory network* (GRN) [Kauffman, 1993]. It is based on distributed computation and local knowledge.

The use of evolution and development to evolve GRNs was investigated in [Bentley, 2004a], also making use of fractals to represent artificial proteins [Bentley, 2004b]. The evolution of GRNs was investigating the potential for evolution to naturally lead to efficiency and graceful degradation in developmental programs. The work showed that such approach creates networks with an ability to function correctly even with missing genes. Evolution would tend to develop over time compact solutions requiring fewer genes to provide a correct solution, and using redundancy within the available genes to better cope with gene removal. Further investigations then assessed the robustness of evolved developmental programs based on fractal proteins [Bentley, 2005b]. Three versions of a program performing the same task were created, one by human design, the second one by genetic programming and the last one by evolution of a developmental program. Damage was inflicted directly on the executable code and experiments showed that only the developmental version could survive damage and show graceful degradation. However, the most robust program could only cope with 0.05% of damage and only 10% of the time. Considering the brittleness of conventional programming language and computer architecture it was a significant result. Yet, while this is an improvement compared to conventional software it is dwarfed by the robustness of natural systems.

Genetic programming (GP) [Koza, 1992, Langdon and Poli, 2002] was used in [Miller and Banzhaf, 2003] for growing developmental computer programs able to regenerate when undergoing damage. The motivation for involving development was based on the self-repair ability induced by development (e.g. organisms) [Miller and Banzhaf, 2003]. The form of evolution the authors used was driven by Cartesian Genetic Programming (CGP), a form of GP that encodes a graph representation of a computer program [Miller and Thomson, 2000]. In this work programs for artificial cells were evolved and the
cells would then develop into a graph of cells where the cell behaviour would depend upon its environment. Following the principles of development all cells would have the same code and they would specialise and develop depending on their environment. On a two dimensional cellular map, where each cell contains the same program, French flag patterns [Wolpert et al., 2006] were grown over time. One aim was to have such developing program always recognisable as a French flag and some experiments consisted in damaging the embryonic map in various ways in order to assess the ability of the program to cope with damage and regenerate itself. While the regenerated flags were very approximate the programs yet showed good ability to cope with damage and grow back missing parts.

Similarly [Haroun Mahdavi and Bentley, 2003] evolved with a standard genetic algorithm [Holland, 1975] a finite state machine to determine the muscle activation sequences in an artificial robot snake. Each solution was a string of bits defining an alternation of commands sent to the muscles and jumps to the next command entry point. Repeating patterns could thus be easily created by jumping back at sequences previously executed. The algorithm generated a population of solutions, sent the solutions to the robot snake control unit for evaluation, evolved a new generation, and so forth. The evaluation function was the distance the snake could travel in the forward direction. During an experiment, after development of a moving strategy, a snake muscle snapped, and the experiment then showed that the evolved control mechanism enabled self-adaptation and redeveloped a new strategy of motion coping with the lost muscle. This allowed the snake to recover up to 85% of its mobility with only 75% of its muscles functioning. Therefore this research showed that using evolution for the control of an artificial robot snake enabled it with fault-tolerance and self-adaptation as the robot was able to cope with the broken muscle and redevelop an efficient moving strategy in spite of the failing part, initially crucial in the previous motion strategy.

**Distributed Knowledge Self-organisation in Artificial Neural Networks**

Another popular field inspired from biology is the field of *artificial neural networks* (ANN). In the 1930s, biophysicists were investigating mathematical approach of neural activities [Abraham, 2002] but a significant breakthrough for both neuroscience and computation was the *McCulloh-Pitts model* presented in [McCulloch and Pitts, 1943]. The novelty in this work was to use logic and computation, in the form recently introduced by Turing [Turing, 1936], to understand neural mechanisms. This model tends to be considered as the first formal artificial neural network. Any Boolean function could be implemented with this model and it was shown to be Turing complete [Kleene, 1956]. However these networks were fixed and a new network would be required for each new problem.
The next significant step was the introduction of the perceptron [Rosenblatt, 1958]. The main difference was that input signals are weighted (i.e. multiplied by a weight factor). The perceptron was able to show adaptive capabilities based on a supervised reinforcement learning method, modifying the weights, and to perform for instance pattern recognition [Rennard, 2006]. A similar approach was known as the Adaline (Adaptive Linear Neuron, and later Adaptive Linear Element) [Widrow and Hoff, 1960] and its learning method, the delta rule, was based on a gradient descent technique by minimising the quadratic error. The weight update rule for the weights of a given neuron is given by Equation 2.3

$$w_{i,k}(t+1) = w_{i,k}(t) + \lambda \cdot (d_i - y_i) \cdot x_{i,k}$$  \hspace{1cm} (2.3)

where $i$ is the processed neuron index, $k$ is the neuron input index, $w_{i,k}$ is the $k^{th}$ weight of neuron $i$, $t$ is a time instant, $\lambda$ is a small constant called the learning factor, $d_i$ is the desired output of neuron $i$, $y_i$ is the current output of neuron $i$, and $x_{i,k}$ is the $k^{th}$ input value of neuron $i$. These two approaches were however still very limited as they could only perform linear separation [Minsky and Papert, 1969]. The field of artificial neural networks has then been discarded for about ten years because of its current limitations at the time and the success of other approaches such as symbolic artificial intelligence [Rennard, 2006].

However, the introduction of feed-forward multi-layer perceptrons and the error backpropagation algorithm [Rumelhart et al., 1986, Werbos, 1974], also based on a gradient descent technique and presented in Algorithm 2.2, allowed neural networks to solve non-linear problems. A common and trivial example of non-linearity that multi-layer networks can solve and single neurons cannot is the X-Or problem [Rennard, 2006, Yanling et al., 2002]. The X-Or gate indeed requires two neurons at least. While such trivial problem can be solved by setting up the weights manually [Yanling et al., 2002], a backpropagation algorithm is able to find appropriate weights for a two or more neurons structure (as shown in Chapter 7) and thus demonstrates ability to solve non-linear problems. One of the first significant application of the backpropagation algorithm was NETtalk [Sejnowski and Rosenberg, 1987], a multi-layer perceptron able to pronounce English text. Such networks can also be used for pattern recognition, function mapping, constraints optimisation, image segmentation [Tang et al., 2007] or image compression [Abdel-Wahhab and Fahmy, 1997]. Despite its broad use, the backpropagation algorithm suffers major limitations. It is slow to converge for large networks [Hinton, 1989] and there is no a priori knowledge regarding the neural structure to be used for a given problem. Also while this method does guarantee a local minimum it does not guarantee a global minimum [Minsky and Papert, 1969], which is a clear limitation [Frasconi et al., 1993]. There are many other limitations and potential methods to overcome some of them [Rennard, 2006] but exploring them is beyond the scope of this chapter.
Algorithm 2.2 Gradient backpropagation algorithm for feed-forward networks. \( i \) is the processed neuron index, \( k \) is the neuron input index, \( w_{i,k} \) is the \( k^{th} \) weight of neuron \( i \), \( h_i \) is the weighted sum of the inputs of neuron \( i \), \( d_i \) is the desired output of neuron \( i \), \( y_i \) is the current output of neuron \( i \), \( x_{i,k} \) is the \( k^{th} \) input value of neuron \( i \), \( e_i \) is the current error of neuron \( i \), \( \kappa_i \) is the set of inputs of neuron \( i \), and \( \varphi_i \) is the set of neurons neuron \( i \) fires into.

Random initialisation of all inter-neuron connection weights \( w \)

repeat
   Pick up a learning example in the training set and process it through the network
   for all neurons \( i \) do
      if \( i \) is an output neuron then
         \( e_i = g'(h_i) \cdot (d_i - y_i) \)
      else
         \( e_i = g'(h_i) \cdot \sum_{j \in \varphi_i} w_{j,i} \cdot e_j \)
      end if
   for all inputs \( k \in \kappa_i \) do
      \( \Delta w_{i,k} = \lambda \cdot e_i \cdot x_{i,k} \)
   end for
   end for
until end condition (e.g. error below a given threshold, maximum number of learnt examples reached)

While ANN are a bio-inspired method, approaches such as feed-forward networks remain far from the actual brain structure [Kandel et al., 1991]. More biologically plausible neural structures have been investigated, such as Hopfield networks [Hopfield, 1982]. This model, using recurrent connections, is based upon auto-associative memory which resembles the human memory principles. Information is no longer retrieved by its address, like in conventional computer memories, but by partial information. Hopfield demonstrated that his network converges towards a local minima, and used a Hebbian learning [Hebb, 1949] to build the auto-associative memory. These networks present strong analogies with the brain as they are able to identify noisy or partial data but they can also create false memory or forget data [Rennard, 2006].

Another sort of neural networks is known as self-organising networks. One of such networks is known as self-organising maps, or Kohonen maps [Kohonen, 1982], and is based on unsupervised learning. The work originates in the observation of a topological mapping between the body areas and the associated brain areas [Kandel et al., 1991]. The structure is based upon a single layer of neurons, apart from the inputs, and considers neighbourhood in its learning process. The aim is to attribute a single output neuron to inputs with similar characteristics, but to also attribute neighbour neurons to similar characteristics. The self-organisation property of the Kohonen map makes it a computational method for data classification with non-linear ability where the principal component analysis only performs linear transformations [Blayo and Demartines, 1991]. It can
also be used to build semantic maps (e.g. classification of animals in groups based upon characteristics) or functions approximation [Renard, 2006]. Another self-organising model is known as adaptive resonance theory networks (ART networks) [Grossberg, 1976]. It is an unsupervised learning model that attempts to recognise the input data in its memory, based on a vigilance parameter, and create additional neurons to handle this input if the data is not recognised. These networks keep the acquired knowledge but can also expand, thus self-organise, to learn additional data. They mainly address problems such as pattern recognition, classification and prediction.

Later, in Chapter 7, this thesis investigates how to create flexible neural structures based on asynchrony and local knowledge using systemic computation.

Artificial neural networks are therefore a valuable approach to learning based on distributed computation and where the knowledge is self-organised across the network. They are able to perform non-trivial tasks (e.g. English text pronunciation, data reconstruction, prediction) out of the massive interaction of simple interconnected elements potentially with a learning ability. Whether using backpropagation or Hebbian learning neural networks show self-adaptation by adjusting their inner weights to solve a given task. Also, structural self-organisation and autonomy properties are demonstrated within Kohonen maps, ART networks or some developmental neural structures [Khan et al., 2007, 2008]. However, there is no ad hoc solution to provide the appropriate neural structure to a given problem for most of the networks, especially the ones using the backpropagation algorithm. This issue can be addressed by evolving neural networks [Mitchell, 1996, Renard, 2006]. Evolutionary algorithms are indeed well suited to large search space problems (here the infinite possible neural structures) as previously discussed.

**Self-organisation and Self-adaptation within Swarm Intelligence**

Considering civilisations of living organisms, insects such as ants are yet another example of complex overall behaviour resulting from individual interactions [Bentley, 2002, Renard, 2002]. The field of swarm intelligence [Beni and Wang, 1989] mimics this concept in algorithms such as ant colony optimisation [Dorigo and Stützle, 2004] or particle swarm optimisation [Kennedy and Eberhart, 1995], based on collective behaviour of decentralised (i.e. distributed and local-knowledge based) and self-organised systems.

In an ant civilisation, the queen (except for the rare queen-less colonies) produces individuals from all castes (e.g. males, workers) where each individual fulfils a specific role with as sole final aim the benefit of the colony. The distribution of the work is horizontal (i.e. no hierarchy). No ant gives or receives orders, they are agents performing simple tasks and communicating with each other through the release of pheromones. Workers
carry out the daily tasks of the colony such as foraging, brood and queen caring, anthill repair and defence without a centralised unit distributing the tasks [Passera and Aron, 2005]. It is a remarkable example of self-organisation.

Ant colony simulations [Dorigo and Stützle, 2004] rely on a simplified view of the ant workers behaviour to create algorithms for the exploration of large search spaces. In this technique artificial ants evolve through an environment representing the search space. Each ant has a limited potential of decision (e.g. go forward, forward left or forward right, turn round, or stay here) and move through the environment in a continuous manner, as opposed to previous approaches such as genetic algorithms where solutions could jump from one position in the space to another after undergoing for instance a recombination. Similarly to real ants, artificial ants, starting from the anthill, randomly explore the search space and upon finding a food source return to the colony laying down pheromones along their path. To return to the colony, real ants keep track of directions and distance travelled [Sommer and Wehner, 2004]. Artificial ants simplify this by being given a sense of direction to get back to the colony. Pheromones are the communication vector between ants and as such they attract ants. Therefore the presence of pheromones in the neighbourhood of an ant impacts its movement decision, which otherwise is random. The more pheromones on a trail, the more an ant is attracted to it. Consequently ants randomly hitting a pheromone trail will most likely follow it, find the food and get back to the colony reinforcing the pheromone trail, which gets even more attractive to ants, and so forth. This is the principle of stigmergy [Grassé, 1959]. The environment is modified and individuals interact with each other through their environment rather than directly. Also, pheromones evaporate over time. Hence a long path between two points should have less pheromones over time than a short path between these two same points, making ants good at finding shortest path between points. Algorithm 2.3 summarises an ant simulation algorithm as presented in [Dorigo and Gambardella, 1997] for solving the travelling salesman problem. From these observations was founded the ant colony optimisation (ACO) approach [Dorigo et al., 1999]. It has been tested against combinatorial optimisation problems such as the travelling salesman problem (TSP) and has been shown to outperform other techniques such as evolutionary algorithms or simulated annealing [Dorigo and Gambardella, 1997, Dorigo et al., 1999]. In the TSP the task is, given a list of cities and their respective pairwise distances, to find the shortest tour that visits each city exactly once [Schrijver, 2005]. Since then this approach has been used in network routing [Kassabalis et al., 2001, Ward, 1998], data analysis or graph partitioning [Bonabeau et al., 1999]. Ant algorithms thus present a partially stochastic, distributed, parallel and asynchronous (e.g. ants are independent from one another) approach. They display flexible (responding to internal perturbations and external challenges), robust (the failure of some individual does not
Algorithm 2.3 Ant simulation algorithm working on a graph as presented in [Dorigo and Gambardella, 1997] for solving the travelling salesman problem.

Initialise pheromone trails and best found solution
repeat
  Position ants on a starting node
  repeat
    for all ants do
      Ant applies a pseudo-random state transition rule to incrementally build a solution
      Local pheromone update
    end for
  until all ants have built a complete solution
  Global pheromone update
until end condition (e.g. best found solution is satisfying, maximum iterations reached)

prevent the completion of the task) and self-organising (stigmergy) behaviour, making them well suited for distributed problems within a dynamically changing environment [Dorigo et al., 2000].

Another swarm algorithm is known as particle swarm optimisation (PSO) [Kennedy and Eberhart, 1995]. It is partly inspired by the movement of flocks of birds seemingly having a specific group behaviour but where all individuals simply follow their trajectory and the group with no central decision process [Heppner and Grenander, 1990]. Like with ACO a population of individuals evolves through an environment (the search space of a problem). However each individual can remember its best location and is also aware of the best known location within the population. Individuals tend to move towards their own best known location (first rule) but also towards the global best known location (second rule). Parameters can weight the importance of each rule. Algorithm 2.4 presents a simple PSO algorithm. A stronger importance given to the individually best known location tends to dissolve the group and explore various areas whereas a stronger importance given to the global best known point tend to unify the group and converge to a local maximum. However, as in the aforementioned methods there is no rule for setting up these parameters. Like with ACO algorithms PSO approaches explore the space continuously. Applications of this method involved artificial neural networks training or optimisation problems in mechanics or biology [Eberhart et al., 2001].

Swarm Intelligence therefore demonstrated to be another valuable nature-inspired approach to solve complex problems, and particularly optimisation problems, by using stochastic, distributed, parallel, and asynchronous. They display self-organising behaviour through pheromone laying and creation of tracks between the colony and the food. They also display self-adaptive behaviour by finding a new path if a track is


**Algorithm 2.4** Particle swarm optimisation algorithm

Initialise all particles at random location  
Initialise best particles and global fitness to lowest value  
repeat  
  for all particles do  
    Calculate fitness value  
    if fitness value is better than best particle fitness value then  
      Update best particle fitness value  
    end if  
  end for  
Update best global fitness value  
for all particles do  
  Update velocity using the global, particle and neighbourhood best locations, with each value weighted by a parameter and a random number  
  Update location using the velocity value  
end for  
until end condition (e.g. maximum number of iterations reached)

damaged. Such algorithms are thus robust to unexpected conditions and show fault-tolerance and self-repair abilities within the search space environment (i.e. not at the software memory fault or hardware failure level).

**Self-organisation and Self-adaptation for Detection in Artificial Immune Systems**

More recently artificial immune systems (AIS) algorithms, inspired by the principles, structure and function of the immune system, were developed as adaptive systems with capacities of learning and memory designed for solving problems which present data never encountered before [de Castro and Timmis, 2002].

Biological immune systems protect their organism from disease by detecting and killing pathogens and tumour cells. Two types of responses can step in upon the entrance of a pathogen within the organism: the innate response and the adaptive response [Janeway, 2004]. The innate immune system provides an immediate but non-specific response whereas the adaptive immune system provides a pathogen-specific response based upon the memory of known pathogens via the white blood B and T cells. The first exposition to a pathogen provokes a slower adaptive response than succeeding ones as appropriate antibodies have yet to be discovered.

Immune systems display features that can prove useful for computation such as pattern recognition, feature extraction, learning, memory, and noise tolerance. However, the principles behind immunity are in some cases not well understood, even by immunologists, and the field of AIS thus sometimes work with plausible but not proven theories.
such as the danger theory [Matzinger, 1994]. In that respect AIS can also be useful for a better understanding of the biological immune system.

Most of the work conducted in AIS has been based on the adaptive response. Clonal selection algorithms are inspired by the clonal selection theory of acquired immunity where only the cells that recognise the antigens can proliferate [de Castro and Von Zuben, 2002]. They have been designed for machine learning and pattern recognition tasks and showed some valuable capabilities. Negative selection algorithms were inspired by the negative selection process T cells undergo in the thymus and in which T cells that recognise self proteins are destroyed and not allowed to leave [Forrest et al., 1994]. Such algorithms are well suited for problems like anomaly detection where knowledge for positive selection is provided and then identification of non-self results from the non-recognition of self. The approach was initially tested for computer virus detection. Immune network algorithms are yet another class relying on the immune network theory [Jerne, 1974, Perelson, 1989] which suggests that memory is maintained using a feedback mechanism, and otherwise knowledge is forgotten. It has been used in machine learning for clustering and visualisation of data sets [Timmis et al., 2000]. The dendritic cell algorithm is inspired from the function of dendritic cells whose role is to present antigen to other cells of the immune system [Greensmith et al., 2005]. They provide a link between the innate and adaptive immune systems by initiating and shaping the adaptive immune response. The algorithm was first designed for anomaly detection and is presented in Algorithm 2.5 (for full implementation details see [Greensmith et al., 2005]).

This thesis later presents in Chapter 8 a novel algorithm based on the innate response and compatible with a potential additional adaptive response algorithm.

AIS algorithms thus present a new approach to potentially distributed stochastic learning systems. Like genetic algorithms they involve a population of antibodies and use mutations to explore various combinations. These algorithms rely on stochastic and distributed computation to demonstrate an autonomous learning process, self-adapting upon the input, in order initially to perform anomaly detection. Applications also include fault diagnosis, computer security, virus detection, machine learning, scheduling, immunised fault-tolerance and optimisation [Stepney et al., 2006].

Distributed Parallel Computation in Cellular Automata for Self-replication and Emergent Patterns Analysis

In the 1940s, the mathematician Ulam was working on crystal growth using a lattice network [Ulam, 1962], a physical model defined on a lattice, thus based on a discrete space-time, and commonly used in condensed-matter physics. He was interested in the dynamics of graphical configurations emerging from simple rules.
**Algorithm 2.5** An artificial immune system algorithm: the dendritic cell algorithm [Greensmith et al., 2005]. The algorithm takes as input a set of data where each item has to be labelled as safe or dangerous.

Initialise a pool of dendritic cells (DCs)

```plaintext
for all data items do
  Randomly pick up a set of DCs in the DC pool
for all picked DCs do
  Add data item to DCs collected antigens list
  Update input signals (PAMPs, safe, danger, inflammatory cytokines) concentrations
  Update output concentrations of co-stimulatory molecules, semi-mature and mature cytokines
  if number of co-stimulatory molecules > given threshold then
    Remove DC from pool and migrate DC
    Create a new DC and add it to the DC pool
  end if
end for
for all migrating DCs do
  if output concentration of semi-mature cytokines > mature cytokines then
    Set DC as semi-mature
  else
    Set DC as mature
  end if
end for
for all data items do
  Calculate the number of semi-mature and mature DCs presenting this item
  if more semi-mature DC then
    Label data as safe
  else
    Label data as dangerous
  end if
end for
```

In the meanwhile, the mathematician von Neumann, his colleague at the Los Alamos National Laboratory, was working on building a self-replicating machine, the *Kinematon*. The idea was to design a machine able to build another machine. Its principle was based on a composite automaton made of three automata: a controller, a constructor and a copier. Given a description this automata was capable of producing any automata.

However von Neumann faced two major issues for the development of his project: the overly complex mechanism of a physical self-replicating machine, and the cost of the materials required for replication.

Ulam’s idea and advices led von Neumann to develop an abstract model of the lattice network, the *cellular automaton* (CA) [Ulam, 1962, von Neumann, 1966]. Von Neumann
was conscious that doing so was getting rid of half of the problem, ruling out any physicalist demonstration and therefore providing only a functionalist approach:

> By axiomatizing [self-reproductive] automata in this manner, one has thrown half of the problem out the window, and it may be the more important half. One has resigned oneself not to explain how these parts are made up of real things, specifically, how these parts are made up of actual elementary particles, or even of higher chemical molecules. [...] we will simply assume that elementary parts with certain properties exist. The question that one can then hope to answer, or at least investigate, is: What principles are involved in organizing these elementary parts into functioning organisms, what are the traits of such organisms, and what are the essential quantitative characteristics of such organism? I will discuss the matter entirely from this limited point of view.

John von Neumann [von Neumann, 1966], p77

Similarly to Ulam’s lattice network, von Neumann’s CA was two-dimensional. Each cell was impacted by its four orthogonal neighbours (known as the von Neumann neighbourhood), as detailed in Algorithm 2.6. Von Neumann applied his Kinematon’s principle to CA and created algorithmically his self-replicating machine known as the universal constructor [von Neumann, 1966], able to self-replicate endlessly within its cellular universe. This machine has 29 states and takes in input a description of self given as a sequence of cells encoding instructions. It uses a construction arm to first replicate the instruction sequence and then to replicate itself. The first full-implementation was published in 1995 [Pesavento, 1995] and is illustrated in Figure 2.8 4. It was however using 32 states instead of the original 29 states.

Research on cellular automaton has been active since its creation. Von Neumann’s CA was then simplified by Codd [Codd, 1968] who recreated the universal constructor with

---

In the end of the 1970s, Langton created his loops [Langton, 1984] based on Codd’s CA but also dropping the universality condition. The investigation was no longer about universality but about the necessary organisation for self-replication only. With this simplification he built a simpler automaton using a significantly lesser amount of cells (8 states). Its principle involved gene cells continuously flowing inside the loop and periodically creating an offshoot arm turning into a daughter loop next to the parent one, as shown in Figure 2.9 4. The loop is a closed data path sheathed with cells in fewer states (8 instead of 29), raising the question of what kind of logical organisation is necessary, rather than effective, for an automaton to be able to universally reproduce itself. Later, this CA was simplified even further by Banks who provided a 4 state universal constructor [Banks, 1971].
Figure 2.9: Langton’s loops at successive iterations with (a) providing the states colour code. (b) shows the loop in its initial state. (c) shows the duplication of genes at the fork between the arm and the loop. (d) shows the growth of the arm. (e) and (f) show the turn left action in the replication process. (g) shows the creation of states 5 and 6, progressing in (h) to initiate the new arms in (i). (j) shows the parent loop replicating in the up direction while the daughter loop replicates on the right. (k) shows the end of a loop that could not replicate on the left and thus died out. (l) shows a loop colony resulting from many replications.

State 2 and containing core cells in state 1. Genetic instructions are sequences of non-zero state followed by state 0 cells. When sequences reach the fork between the loop and the arm, sequences 70 and 40 duplicate and a copy of the sequence flows along the arm while the initial sequence loops (Figure 2.9(c)). The sequence 70 increases
the length of the arm (Figure 2.9(d)), the sequence 40-40 creates a ninety degree left
turn (Figures 2.9(e) and 2.9(f)). State 3 is only used as a temporary state (visible
for instance in Figures 2.9(e), 2.9(f) and 2.9(i)). States 5 and 6 are used when a
daughter loop is done and separating from the parent loop (Figure 2.9(g)). The state 5
cell travels counter-clockwise along the sheath of the parent loop until the next corner
where it initiates the next arm to be built (Figures 2.9(h) and 2.9(i)). The state 6
cell joins the data flow of the daughter cell until the next corner where it initiates the
arm to be built (Figures 2.9(h) and 2.9(i)). Loops attempt to replicate successively
counter-clockwise in each cardinal direction (Figure 2.9(j)). When a loop already exists
at the place another loop intends to replicate, the loop about to replicate terminates and
settles as a wire containing cells in state 1 (Figure 2.9(k)). The loop colony can expand
infinitely in all directions as shown in Figure 2.9(l). The novelty in this construction
was that the information of the program was also the information necessary to build the
program. The information is therefore both itself and a description of itself, thus not
requiring a kinematic structure and suggesting that a significantly simpler model that
von Neumann’s is feasible. Langton’s loops have since then been simplified [Byl, 1989,
Reggia et al., 1993, Stauffer and Sipper, 2001].

The main contribution of this work so far has been to show that self-replication, one
fundamental property of living systems, can be explained in terms of interaction between
simple elements, independently from any physical realisation, as shown in [von Neumann,
1966] or [Langton, 1984]. This self-replicating behaviour also implies that the properties
of autonomy and self-organisation, underlying self-replication, can be achieved in CAs.

However, CA gained most of their popularity when [Gardner, 1970] introduced Conway’s
Game of Life (also known as Life). The game of life is an infinite two-dimensional CA
using a Moore neighbourhood (the 8 surrounding cells of a cell) and where each cell can
be either dead or alive. Even though it has the title of game, it is a zero-player game.
The only interaction with the game is by setting the initial configuration. Then the
rules are the following:

- live cells surrounded by two or three live cells remain alive,
- live cells surrounded by less than two or more than three live cells die,
- dead cells surrounded by three live cells become alive.

With these three rules one could observe some stable structures, oscillating structures,
self-replicating structures, self-moving structures or infinitely growing structures [Levy,
1992, Remnard, 2002, Wolfram, 2002] as shown in Figure 2.10. The game of life was
later proven to be Turing Complete [Rendell, 2002]. Therefore, in spite of its simplicity
Chapter 2 Computational Biomimetism

Figure 2.10: Conway’s game of life. Black and white cells are respectively live and dead cells. (a) and (b) are still patterns. (c) and (d) are two-phase oscillator patterns. (e) and (f) are self-moving patterns in four steps.

the game of life can provide a broad range of emerging behaviours within autonomous self-organising structures running on a distributed and parallel mode of computation. This form of emergence is however qualified as weak emergence [Bedau, 1997], as it can be traced down to its individual constituents.

Wolfram also dedicated a vast part of his research to CA and presented in [Wolfram, 2002] his New Kind of Science, a defence of the empirical approach to the understanding of computational systems such as CA and arguing that the observations made in such computational systems are very relevant to natural systems. This point is illustrated with physical or biological processes such as crystal growth, the breaking of materials, fluid flow, plant growth, or biological pigmentation patterns. Wolfram focussed particularly on one-dimensional cellular automata and suggested a classification of their behaviour within four classes [Wolfram, 1984, 2002]. Class 1 automata evolve to settle within a fixed attractor states. Class 2 automata evolve towards stable configurations or cyclic attractors (i.e. periodic structures). Class 3 automata evolve towards strange attractors (i.e. chaotic patterns such as fractal patterns). Class 4 automata evolve towards complex and unpredictable behaviour. Attractor states can be represented in the phase domain as illustrated in Figure 2.11. The first three classes can be linked to known physical behaviours. A fixed attractor (Figure 2.11(a)) can be seen as the stop state of a system. A cyclic attractor (Figure 2.11(b)) can be illustrated by the movement of a clock, (if discarding friction as otherwise it would also evolve towards a fixed attractor), or the movement of the heart or the breathing principle (as long as the organism is alive). Strange attractors [Ruelle and Takens, 1971] such as Lorenz attractor [Lorenz, 1963] (Figure 2.11(c)) show high sensitivity to initial conditions. Such
Chapter 2 Computational Biomimetism

Figure 2.11: Attractors

behaviour is commonly described as the butterfly effect [Lorenz, 1972]. Lorenz showed that such behaviour rules the atmospheric behaviour [Lorenz, 1963] and as such long term weather forecast is impossible. The last class is however less clearly related to known physical phenomena.

Langton also investigated the behaviour of cellular automata to study artificial life [Langton, 1986] and used a parametric approach [Langton, 1990] with parameter $\lambda$. Considering an automaton with $s$ states using a neighbourhood of $c$ cells, there are $s^c$ possible neighbourhood configurations. Amongst all the possible states of the automaton, one state (e.g. state 0) is designated as the quiescent state. When changing the CA’s state, the transition function can therefore lead to the quiescent state or any of the remaining states. With $t_q$ the number of transitions towards the quiescent state, the parameter $\lambda$ is defined by:

$$\lambda = \frac{s^c - t_q}{s^c}$$

$\lambda$ is a measure of probability that a transition leads to an active state. As a result, $\lambda = 0$ means that all configurations lead to the quiescent state, and $\lambda = 1$ means that no configuration leads to the quiescent state. By varying the value of $\lambda$ the four classes of Wolfram can here be also identified. However in this classification class 4 is reached before class 3. The classification of behaviours is, by increasing the value of $\lambda$ from zero to one: fixed, cyclic, complex and chaotic. This work, as well as the work from Wolfram, highlighted the potential of systems at the edge between order and chaos, suggesting that complex automata are particularly well suited for information processing and that only such systems can be universal. Consequences of these studies are important. They suggest that among all possible universes, our universe is precisely located at the edge of chaos. They also suggest that life itself is originating in such area of behaviour and that evolution was the progressive process by which life was carved into complexity [Langton, 1990].
The work of Wolfram and Langton therefore clearly showed that CAs can display complex behaviour, built out of simple rules, and thus show open-endedness.

Finally Rucker sees in this vision of universal automatism a third revolution, following the Newtonian and Darwinian revolutions, and states that “Everything is a computation” [Rucker, 2005].

Cellular automata can thus be acknowledged as a valuable approach to emergent, parallel, distributed and local computation able to show natural properties, particularly of living systems [Monod, 1970], such as autonomy and self-replication. Beyond an inspiration from nature, their study also provided significant insights regarding the behaviour of our own universe. However they suffer a clear limitation, their fixed structure. Even in a digital physics space [Zuse, 1970] there is no reason to limit the perspective to such a fixed structure (i.e. imposed grid structure and neighbourhood size).

### 2.4.4 Summary

This section presented software approaches to natural computation. Table 2.3 provides for the reviewed software approaches an evaluation of the level of achievement of the natural computational properties, as well as an evaluation of the behavioural properties that the techniques are designed to achieve.

<table>
<thead>
<tr>
<th>Property</th>
<th>Conventional software</th>
<th>Nature-inspired software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recovery blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-version programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-agent systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evolutionary algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artificial neural networks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swarm intelligence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artificial immune systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cellular automata</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stochastic</td>
<td>** * ** *</td>
<td>** * ** *</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Parallel</td>
<td>** ** ** **</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>Continuous</td>
<td>** ** **</td>
<td>** ** ** ** **</td>
</tr>
<tr>
<td>Distributed</td>
<td>*</td>
<td>** ** ** ** **</td>
</tr>
<tr>
<td>Approximate</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Embedded</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Circular causality</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Local knowledge</td>
<td>*  **  *</td>
<td>*  ***</td>
</tr>
<tr>
<td>Self-organised</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Homoeostatic</td>
<td></td>
<td>*  **  *</td>
</tr>
<tr>
<td>Robust</td>
<td>*  *  *</td>
<td>*  **  *</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td>*  *  *</td>
<td>*  **  *</td>
</tr>
<tr>
<td>Autonomous</td>
<td>** ** **</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>Open-ended</td>
<td></td>
<td>*  **  *</td>
</tr>
<tr>
<td>Complex</td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>
While conventional software approaches are not well suited for the simulation of such properties, nature-inspired software approaches have shown a clear potential. Nature has indeed been a rich source of inspiration for the development of new kinds of algorithms. These algorithms provided new exploration, optimisation or development techniques showing good abilities to deal with non-linear problems in potentially noisy environments [Ashlock, 2005]. Properties such as autonomy, robustness or self-organisation found in biological systems could potentially emerge from stochastic, distributed, local knowledge-based, parallel and/or asynchronous computation. Such properties can be of great interest to fields like software robustness but also data classification or problem optimisation environments. Approaches based on evolution and/or development proved to be able to yield more robust software, yet their potential remains limited as the most robust program presented in [Bentley, 2005b] was only able to cope with 0.05% of damage. The weakness of such approaches seems to mainly lie in the brittleness of traditional processors and operating systems. Despite its distributed structure, a genetic algorithm remains as brittle as the operating system it runs from. A failure of a memory chunk might terminate the whole system while a physically distributed system would only lose a sub-component. The methods presented in this section therefore largely ignore the nature of computation itself. Natural systems are based upon a radically different computational approach, as seen in Table 2.1 (page 51), and as a result their implementation on conventional machines is limited by the conventional properties (e.g. brittleness in the previous example) these machines impose. Without tackling the underlying architecture, it is clear that the underlying incompatibilities of conventional computation cause significant problems to provide sufficient flexibility to enable the desired natural characteristics such as fault-tolerance or self-repair. Alternative hardware platforms have been discussed in the previous section and can provide native support for some natural properties. As alternative hardware also suggests alternative way to perform computation (e.g. nature of computation, software engineering, programming) the next section discusses alternative computational paradigms. It then reviews the early work conducted on systemic computation which was introduced to unify the notions of electronic and natural computations within a single paradigm, thus providing both a hardware architecture and a computation paradigm.
2.5 Alternative Paradigms of Computation

2.5.1 Introduction

To help model and understand natural processes, these processes need to be understood in terms of computation. In that respect traditional paradigms like conventional languages (e.g. procedural languages, object-oriented languages), designed for conventional computation, are inappropriate to define natural computation. Alternative methodologies for the analysis of computation have been developed to tackle this issue regarding complex or natural systems.

2.5.2 Paradigms Originating in Mathematics, Physics or Technology

Constrained Generating Procedures

Generalisations such as constrained generating procedures (CGPs) suggest a method of design breaking down complex systems into mechanisms and constraints of interactions [Holland, 1998]. CGPs are finite state machines based on a systemics view of reduction that enable analysis of the complexities and emerging behaviours within computational phenomena. Holland explains their principle as follows:

*The models that result are dynamic, hence procedures; the mechanisms that underpin the model generate the dynamic behavior; and the allowed interactions between the mechanisms constrain the possibilities, in the way that the rules of a game constrain the possible board configurations.*

John Holland [Holland, 1998], p26

CPGs can be summarised as follows:

- The rules of a CGP are its mechanisms.
- Each mechanism has a state and receives inputs.
- A network of mechanism linked together is a CGP.
- All the possible state configurations of all mechanisms define the various potential global states of a CGP.
- CGP can be combined to form bigger CGP, thus defining embedded hierarchies.
Let $S$ and $I$ respectively be the set of possible states and the set of possible inputs of a given mechanism. The transition function changing the state of a mechanism thus has the form

$$f : I \times S \rightarrow S$$

Over time, by introducing the time instant variable $t$, the current state of a mechanism $s \in S$, and its current input $i \in I$, this becomes

$$s(t + 1) = f(i(t), s(t))$$

The set of mechanisms $F$ in a CGP are called primitives. Everything is thus constructed from these primitives. Two states are connected if the state of one mechanism determines an input of the other mechanism. As a result a CPG is a network of mechanisms from $F$, each mechanism being potentially instantiated several times, that are connected (e.g. a neural network). Inputs that are not determined by other mechanisms are free inputs. Interaction with the outworld can be done by providing values for these free inputs.

To then use CGP to analyse the implication of each mechanism, or constituting CGP, an incremental approach based on the CGP embedded hierarchy can be performed. The possible situations are:

- A CGP can be a single mechanism.
- Provided a CGP $C$ with a free input, connecting a mechanism of $C$ to this input produces a new CGP $C'$.
- Provided a CGP $C_1$ with a free input, a second CPG $C_2$ can be connected to $C_1$ through this input which produces a third CGP $C_{12}$.

All CGPs based on the primitives $F$ can be built using these three steps.

CGPs have been largely illustrated with game theory but also highlighting the potential relevance their study can provide for real world systems such as war strategy. Holland also discussed how to build cellular automata, genetic algorithms or general purpose computers with CGPs. For Holland, the study of emergence is closely tied to the generation of large and complicated domains out of a few laws. In this respect, an underlying purpose for such language is the study of emergence itself rather than given examples of emergence. He argues that such study should help us in the future to predict systems such as economies, businesses, ecosystems, life or consciousness. However, while CGPs provide a way to model increasingly complex systems from basic to composite CGPs level, it does not provide a unified theory defining emergent properties in terms
of CGPs. Combining CGPs can yield more complex CPGs, yet this does not guarantee an emergent behaviour.

**π-calculi**

Mathematical models such as the π-calculus [Milner, 1991] have been used or adapted for the simulation of biological systems [Phillips and Cardelli, 2004a, Regev et al., 2001]. The π-calculus is a process calculus that was initially designed to describe and analyse properties of concurrent computation in systems such as mobile telephones. Considering processes $P$ and $Q$, a communication channel $c$, and names $x$ and $y$ (names define channels or values), the syntax is the following:

- Concurrency $P | Q \quad P$ and $Q$ are executed in parallel
- Input communication $c(x) \cdot P \quad P$ waits for a message on $c$ and binds it to $x$
- Output communication $\overline{c} < y > \cdot P \quad y$ is emitted on $c$ before running $P$
- Replication $!P \quad P$ can replicate infinitely
- Restriction $(\nu x)P \quad$ allocates a new name to $P$ only
- Nil process (or inactivity) $0$
- Equivalence $P \equiv Q$
- Reduction $P \rightarrow Q \quad P$ becomes $Q$

[Regev et al., 2001] presents an approach using the π-calculus for the modelling of biochemical networks and applying it to the RTK-MAPK signal transduction pathway [Heldin and Purton, 1996, Lewis et al., 1998]. However, the absence of stochasticity in the π-calculus makes the MAPK model very approximate.

The π-calculus has been extended to several variations such as the asynchronous π-calculus [Boudol, 1992] involving asynchronous message passing, or the stochastic π-calculus [Priami, 1995]. The stochastic π-calculus addresses the stochasticity issue by providing rates for communication actions. The occurrence of processes is dictated by a stochastic variable, the rate, defining the number of times the process is expected to happen per time unit. The rate is typically drawn from an exponential distribution and is memoryless (i.e. the probability that a process occurs is independent of the number of times it occurred in the past). With this rule, processes are in a race condition: the fastest activity only succeeds. The duration being a stochastic variable the outcome of a race can be different each time. Also each new race is independent from previous races.
The notion of rate leads to the notion of probability for communications to occur, which is an essential feature for an accurate modelling of natural systems.

To correctly model but also implement biological processes, [Phillips and Cardelli, 2004a] introduced a variation of the stochastic π-calculus along with an execution environment for it. This version uses a delay to model the rate. This calculus has later been extended to include graphical output. In the following example using the additional process $R$

$$P \rightarrow \tau \cdot Q | c(x) \cdot R$$

$P$ would reduce to $Q$ if the delay $\tau$ elapses before a message is received on channel $c$, and would reduce to $R$ otherwise. (This example uses the standard π-calculus notation while the aforementioned methods can use a slightly different notation.) The authors also provided models of biological processes [Phillips and Cardelli, 2005, Phillips et al., 2006] such as a bistable gene network (BGN) as presented in [François and Hakim, 2004] and a mitogen-activated protein kinase (MAPK) cascade as presented in [Huang and Ferrell, 1996].

However, the computer science and communications theory origins of the π-calculus can make the resulting models non-intuitive for expressing interactions and transformations of entities happening in natural systems. In [Phillips and Cardelli, 2004a] the binding model $H + Cl = HCl$ is used as an example to introduce and teach the stochastic π-calculus, as given in Figure 2.12. Its purpose is to model the binding process between hydrogen ($H$) and chloride ($Cl$) ions to form hydrogen chloride ($HCl$), illustrated in Figure 2.12(a). However in the stochastic π-calculus model it is not obvious that $H$ and $Cl$ become bound to each other, as shown in Figure 2.12(b) and Figure 2.12(c) where processes appear to be separate despite their binding and interaction being the purpose of the model.

The stochastic π-calculus approach to modelling biological processes and the $HCl$ model are detailed further and compared to the SC approach in Chapter 3. Also the BGN and MAPK cascade models are later compared with SC implementations in Chapter 9.

**Ambient Calculus**

*Ambient calculus* (or mobile ambients) [Cardelli and Gordon, 1998] was developed to unify the notions of mobile computing and mobile computation. Mobile computing concerns computation happening within mobile devices such as laptops or personal digital assistants whereas mobile computation concerns mobile code moving between computing devices. This method was developed with especially in mind the World Wide Web that required, by its distributed nature, mobile computation.
Figure 2.12: (a) provides a 3D representation of a hydrogen chloride (HCl) molecule clearly showing H and Cl bound together. (b) and (c) respectively provide the graphical representation and the corresponding notation of the stochastic π-calculus binding model $H + Cl \rightleftharpoons HCl$ from [Phillips and Cardelli, 2004b] where the binding of H and Cl is not evident. (Note that the syntax differs from the standard π-calculus notation, see [Phillips and Cardelli, 2004a] for full details about this notation.)

In ambient calculus, objects are defined as *ambients*. Ambients are agents defined as a bounded place where computation happens. The concept of boundary is important as it draws the line between what can be mobile, the agents, and what defines the agents themselves. Examples of ambients include a web page (bounded by a file), a virtual address space (bounded by an addressing range), a Unix file system (bounded within a physical volume), a single data object (bounded by *self*), and a laptop (bounded by its case and data ports). Counter-examples include threads (where the boundary of what is reachable is difficult to determine) or logically related collections of objects [Cardelli and Gordon, 1998]. Ambients can be nested within ambients (i.e. embedded hierarchies). Computation is defined as the movement of ambients (e.g. enter or leave an ambient), and is asynchronous.

In ambient calculus, locations are abstracted and represented by a topology of boundaries. The mobility of process is defined as the crossing of boundaries, as opposed to passing process names over channels like in the π-calculus. The concept of security is represented as the ability or inability to cross boundaries. Interactions between processes happen within a shared location [Cardelli and Gordon, 1998].

Ambient calculus, just like π-calculus, possesses primitives for parallel execution (concurrency), restriction, inactivity and replication. The new additions are the ambients and the exercise of capabilities. Ambients (i.e. boundaries) have a name $n$ and a process $P$ running inside. Such ambient is noted $n[P]$. $P$ can be a parallel composition of processes and processes run whether the surrounding ambient is moving or not. Several
ambients can have the same name, which enables the replication of ambients (e.g. services). Also, actions affecting the ambients hierarchy can be performed. The concept of capabilities controls the feasibility of these actions which can be allowed or not by ambients onto other ambients. These actions can be either entering an ambient, leaving an ambient or opening an ambient, as given below with the notation:

- **Entry** \( \text{in } m.P \) instructs the surrounding ambient to enter a sibling ambient called \( m \), choosing one randomly if several such ambients exist,

- **Exit** \( \text{out } m.P \) instructs the surrounding ambient to exit its parent ambient called \( m \),

- **Open** \( \text{open } m.P \) instructs the surrounding ambient to dissolve the boundary of a sibling ambient called \( m \).

Such calculus therefore presents an approach more suitable for natural processes with structure and organisation being at the core of the paradigm through the concept of ambients. Ambients can enter or leave, thus modify hierarchies, themselves protected by capabilities. However this algebra was designed for mobile processes and was not concerned with the modelling of natural processes. Movement can be initiated by a process and involves another process but this one-sided manner does not reflect the concept of reciprocal interaction between two processes, as found in natural processes. In this respect [Regev et al., 2004] highlighted the inadequacy of mobile ambients for the modelling of biological processes and introduced *BioAmbients*, which will be reviewed further below.

**Petri Nets**

In 1939, in order to describe chemical processes Carl Adam Petri invented at the age of 13 the *Petri nets* [Petri, 1966]. They are a graphical tool with a corresponding mathematical theory to describe and analyse concurrent processes. The processes are described using *places*, *transitions*, and *directed arcs*, and can involve the notions of choice, iteration, and concurrent execution. Arcs run between places and transition. An arc going from a place to a transition is called an *input arc* whereas an arc leaving a transition towards a place is called an *output arc*. Places can contain *tokens*. Transitions can fire if there are tokens available at the end of the input arc, in which case these tokens are taken to the place reached by the output arm. Figure 2.13 provides a Petri net model of a sequence of chemical reactions and Figure 2.14 provides an exclusive or (X-or) Boolean operator Petri net model. Note that in these examples nothing happens once the chemical reaction is over or once the X-or computed, but Petri nets can be cyclic.
An important feature in Petri nets is their non-determinism. When transitions are enabled simultaneously there is no pre-defined order for their execution. Such feature makes Petri nets well suited for the modelling of concurrent behaviour in distributed systems.
Petri nets and derived tools have also been used for the modelling of biological systems [Chaouiya, 2007, Peleg et al., 2005], representing and simulating them by breaking them down into sub-processes that may be described at various levels of details. However, the lack of a single formalism for the various simulation tools potentially lead simulations of a same system on different platforms to different results. They also cope badly with systems requiring many states and are thus only appropriate for modelling systems with a limited amount of states [Peleg et al., 2005]. For biological systems such as long genetic sequences this is a significant limitation.

**Statecharts**

*Statecharts* [Harel, 1987], or *state diagrams*, are used to describe the behaviour of computer systems. They mainly represent the potential states of systems and the conditional transitions between these states. The described systems are required to have a finite amount of states and statecharts can thus be used to graphically represent finite state machines, as illustrated in Figure 2.15 with a simple switch model.

![Figure 2.15: Model of a simple switch using a statechart. The model involves two states, ON and OFF, with two transitions, switch on and switch off. Each state can also display the entry actions that leads to it.](image)

As complex systems might require a large amount of states to describe their full behaviour, a less naive approach would define these systems with using fewer states. [Harel, 1987] states that this requires the notions of clustering, orthogonality and refinement and thus suggests statecharts as an attempt to address these issues. Figure 2.16 illustrates the clustering of states with a three states statechart. The first statechart shows a regular three states model which is then clustered using Harel’s notation to allow a zoom out on it, hiding with distance more and more details. This method thus allows to simplify the model and limit the amount of states when looking at it from the distance. Details can reappear when zooming onto the model.

This method has been used to design communication protocols [Bar-Tur et al., 1986] and for hardware description [Drusinsky and Harel, 1989]. However statecharts are designed to graphically represent behaviour or data flow and thus do not provide a computational paradigm.
**Figure 2.16:** (a) shows a statechart with 3 states in its regular form. (b) shows the same model using Harel’s clustering notation to put together states A and B in a state D and only show one transition $\beta$. (c) then shows the same model at a coarser level of detail where states A and B are hidden behind super-state D.

**Bigraphs**

As raised by [Regev et al., 2004], a graphical component present at the core of the method, like in statecharts [Harel, 1987], can be adequate to represent biological models.

*Bigraphs* [Milner, 2001] provide a model exploiting topographical and communication ideas pervasive in calculi such as the ambient calculus offering mobile locality, and the $\pi$-calculus offering mobile connectivity. The approach presents a graphical formalism for concurrency within autonomous agents along with rigorously defined mathematical basis and a dedicated theory. A bigraph contains *nodes* and *links* (or *edges*). As can be seen on Figure 2.17 showing a simplified uninterpreted bigraph, bigraphs involve two kinds of structure, hence the name *bi-graph*. The *topograph* offers embedded hierarchies with nodes nested within other nodes and possibilities to reconfigure the space, like in ambient calculus. The *monograph* provides a linked structure by connecting nodes through edges. Nodes have *ports* where connections can be made. Links may fork and

---

5Reproduced with permission from [Milner, 2005].
are independent from locality, offering reconfigurable connectivity like in the $\pi$-calculus [Milner, 2001]. This approach suggests that where you are need not affect whom you can talk to [Milner, 2006].

Figure 2.18 provides an example, taken from [Milner, 2005], that models agents and objects interrelated and interacting. Nodes are assigned a control which defines the kind of node they are (i.e. their role) and their arity (i.e. the number of ports). $B$ represents a building where there are rooms $R$. Agents $A$ (here people with mobile phones) can move within the building and rooms. Computers $C$ are connected together within the building. All the people in the building and people outside the building are connected together indicating they are all communicating (e.g. conference call). Their link is said to be open as it goes outside this graph and can be connected to other host graphs (e.g. involving more agents). An agent in a room has access to a computer. All computers are connected within a building through a network, built within the building, hence the connection to it. The connection from an agent to a computer is a closed link. Also, the bigraph $G$ is divided in two regions (areas delimited by the dotted rectangles). The number of region of a graph is called the width of the graph. The host model $H$ is here contextual, first because it has two holes represented by grey squares where a two regions bigraph could fit, and also because it has two inner names, $y$ and $z$, that can connect to

Figure 2.18: Bigraph modelling interacting agents in a structured environment. (a) provides a partial system that can be composed with a host, given in (b), to produce a full model, as shown in (c).
an inhabitant graph through its *outer names*. Inserting $G$ into $H$ is noted $H \circ G$. This graph still has one open link $x$ and can inhabit another host bigraph.

The textual notation associated with the graphical notation reflects the double structure of the graphs. A *concrete* bigraph $G$ is composed of a place graph $G^P$ and a link graph $G^L$:

$$G = (V, E, ctrl, G^P, G^L)$$

with $V$ the set of nodes, $E$ the set of edges and $ctrl$ the control map assigning a control to each node. A place graph is defined as

$$G^P = (V, ctrl, prnt) : m \rightarrow n$$

where $prnt$ is a parent map, $m$ its inner width and $n$ its outer width. A link graph is defined as

$$G^L = (V, E, ctrl, link) : X \rightarrow Y$$

where $link$ is a link map, $X$ the set of inner names and $Y$ the set of outer names.

The duality between graphical and algebra aspects of bigraphs makes possible for non calculi specialists the modelling of systems within a rigorous mathematical environment. With respect to other paradigms, bigraphs have been shown to inform $\pi$-calculus, ambient calculus and Petri nets [Milner, 2005]. Also, *systemic computation*, investigated in this thesis, can be regarded as a form of bigraph model [Bentley, 2007a].

**Ons Algebra**

Recent work at the crossing of *pregeometry* [Misner et al., 1973], theoretical physics, and algebra, investigated a new algebra called *Ons algebra* [Goertzel, 2007]. Pregeometry, with respect to physics, defines a structure from which geometry develops. It investigates rules that lie deeper than the notion and properties of space, and thus underlie the rules of physics. The idea with Ons algebra is to propose a new form of *Brownian algebra* (a minimalist formalism equivalent to Boolean algebra) [Spencer-Brown, 1969] defining a pregeometry that can give rise to *Clifford algebra* which in turn gives rise to the core of modern physics [Dixon, 1994].

Ons algebra defines an elementary particle called $On$ and noted $N$. Ons and expressions can be bounded by hard and soft boundaries. Soft boundaries noted $<>$ denote boundaries through which entities can easily pass, as opposed to hard boundaries, noted $[\ ]$. The particular construct $[N]$ is called a Chronon and can be noted $C$. The absence of something is the void, denoted $V$. From these simple notations Ons algebra provides a set of basic rules:
• \([\,] =<>= V\) a boundary with nothing inside is Void (noted \(V\));

• \(X + Y\) denotes coexistence;

• \(N + N = V\) \(N\) is annihilator within coexistence;

• \([\![X] + [Y]\!]=[[X + Y]]\) within a hard boundary, hard boundaries are associative;

• \(<[C + X] + X >=< V >= V\) within a soft boundary, no hard boundaries associativity, but this rule instead;

• \(X \hat{\lor} Y = \begin{cases} C & \text{if } X \text{ occurs before } Y \\ V & \text{otherwise} \end{cases}\) temporal precedence;

• \(X \star Y = X + Y + X \hat{\lor} Y\) interpenetration.

The rules of Ons algebra were demonstrated to be able to produce Clifford algebra. This reconstruction from the simple pregeometric foundations of Ons algebra thus provides a significant step towards proposing a simple pregeometric formalism from which physics rules could emerge. While focussing on physics this work presents similarities with systemic computation. It is attempting to reach the core rules from which the foundations of physics develop in a similar way that SC aims at providing a computational foundation for natural systems. As this work focusses on low level theoretical physics it is not clear how this would relate to natural computation modelling in terms of what annihilation or interpenetration would mean, or how to represent interactions and transformations at higher levels of abstraction than the elementary particles.

2.5.3 Paradigms Originating in Biology

BioAmbients Calculus

BioAmbients calculus [Regev et al., 2004] was developed upon the motivation of ambient calculus [Cardelli and Gordon, 1998] and prior work with the \(\pi\)-calculus [Priami et al., 2001, Regev et al., 2001] to stress the importance of compartments in biological systems. It is also an adaptation of the stochastic \(\pi\)-calculus [Priami, 1995]. The aim of BioAmbients is to allow the study of biological compartments of different granularities, the movement of molecules between compartments, and the dynamic rearrangement of cellular compartments and molecular complexes [Regev et al., 2004].

The following example, taken from [Regev et al., 2004] and illustrated in Figure 2.19, provides a BioAmbient model for an enzyme-substrate complex involving a reversible single-substrate reaction.
Figure 2.19: Enzyme-substrate complex model using BioAmbients

\[
\text{System} := \text{enzyme[E]} | \ldots | \text{enzyme[E]} | \text{molecule[S]} | \ldots | \text{molecule[S]}
\]

\[
E := \text{accept e_s bind . ES + accept e_p bind . ES}
\]

\[
\text{ES} := \text{expel unbind . E + expel react . E}
\]

\[
S := \text{enter e_s bind . X}
\]

\[
\text{X} := \text{exit unbind . S + exit react . P}
\]

\[
P := \text{enter e_p bind . X}
\]

BioAmbients accepts a syntax similar to ambient calculus. The enzyme is represented by an enzyme ambient running a process \(E\). The substrate and the product are represented by molecule ambients respectively running the processes \(S\) and \(P\). The \textit{enter} and \textit{accept} capabilities on channels \(e_s \text{ bind}\) and on \(e_p \text{ bind}\) respectively represent enzyme-substrate and enzyme-product binding. The enzyme-substrate binding is modelled by the substrate ambient entering into the enzyme ambient. The resulting enzyme-substrate complex is therefore abstracted as a substrate ambient nested inside the enzyme ambient. The reverse reaction is modelled by the \textit{exit} and \textit{expel} capabilities which can occur on the \textit{unbind} channel, then releasing an intact substrate, or on the \textit{react} channel, then releasing a product.

As can be observed with this approach (for instance compared to the stochastic \(\pi\)-calculus approach reviewed above) the nested hierarchies enable a better representation regarding the relationships between elements. The binding between two elements is no longer represented by two completely separate processes (as seen in the HCl model using the stochastic \(\pi\)-calculus) but is abstracted with nested elements. The clarity of modelling remains nevertheless approximative as the binding of two physical elements, say molecules, does not mean one is arbitrarily nested within the other. Therefore in spite of the capabilities of this method to model biological systems, models may remain inaccurate due to paradigm limitations. Also, the organisation and hierarchies of ambients would benefit from a graphical approach inner to the language and able to reveal organisation and information in a dynamic way [Regev et al., 2004], as has been addressed with approaches such as bi-graphs [Milner, 2001].
Membrane Computing

Membrane computing (also referred to as P systems) [Păun, 2004] abstracts the functioning of biological cells to deal with distributed and parallel computing models. Chemicals are modelled by objects, or symbols, and regions by membranes. Objects are compartmented within membranes that can encapsulate other membranes, thus creating embedded hierarchies. A P system has an outer membrane (or skin) which contains the rest of the system. It can also be seen as a tree rooted in the skin membrane. Membranes are governed by rules that define the production of objects, their consumption, how they are passed to other membrane and how they can interact with each other. As objects can be passed through membranes, membranes are permeable. Two object modifiers are provided, in and out, causing the object to be respectively randomly passed to one of the current membranes children and passed out of the current membrane into its parent membrane or a sibling membrane. Membranes can also dissolve. Symbols are represented by letters, and the special symbol δ denotes the dissolution of the containing membrane. Rules are applied in a non-deterministic and massively parallel way. Priorities can however be given to indicate that a rule should be applied as possible before another rule can apply. The application of rules to a system changes the states of the system and a sequence of state transitions is defined as a computation. Contrary to biological systems, here systems can reach a halting state and the final state (i.e. regions and objects within) is the result of computation. The result can also be objects expelled from the membrane into the environment (i.e. the surroundings of the P system).

P systems can be represented graphically. Figure 2.20 provides a graphical representation of a simple P system. In this system, computation starts in membrane 3 as this is the only membrane containing an object, here af. From there, rules can be applied to a and f. The rule \( a \rightarrow ab \) is applied in parallel to the rule \( f \rightarrow ff \) until the rule \( a \rightarrow b\delta \), which dissolves the membrane, is applied. Upon applying this rule the content now belongs to membrane 2 and is subject to its rules. The content is then \( b^{n+1}f^{2^{n+1}} \).
with $n$ the number of parallel computation steps before applying the dissolution rule. (The exponent notation upon a symbol refers to its number of consecutive occurrences: $b^3$ means $bbb$). After the first step in membrane 2, all $b$ have been turned into $d$. At each further step one $e$ is produced from each $d$. Regarding the $f$ symbols, priority is given to the rule $ff \rightarrow f$, thus all pairs of $f$ are merged into one until the rule cannot apply any more, with only one $f$ left. The rule $f \rightarrow \delta$ then applies. The amount of $e$ produced therefore depends on the amount of $f$ pairs available. $2^{n+1}$ pairs were produced, the first step in membrane 2 converts all $n + 1$ $b$ symbols to $d$ symbols and brings down the number of pairs of $f$ to $2^n$. It takes $n$ steps to get from $2^n$ to $1f$, plus one step to apply the final dissolution rule. The rule $d \rightarrow de$ is thus applied $n + 1$ times, each time on all of the $n + 1 d$, therefore producing $(n + 1)^2 e$ symbols which, after dissolution of membrane 2, will be all sent out to the environment by applying the rule $e \rightarrow (e, out)$. The result of this computation is therefore the amount of $e$ symbols given by equation $(n + 1)^2$ where $n$ is the number of steps performed in membrane 3 before dissolution.

P systems have been proven capable of universal computation [Freund et al., 2005]. It was shown that deterministic P systems can be simulated on a Turing machine in polynomial time [Păun and Rozenberg, 2002]. It has also been proved that a non-deterministic P system variant, the P systems with active membrane, can solve the NP-complete boolean satisfiability problem (SAT) in linear time [Păun, 1999]. This variant relies on an enhanced parallelism which allows the membranes to divide exponentially. This raises the problem of a possible practical implementation [Păun, 1999] as the physical potential counterparts of membranes (i.e. processors) cannot easily be replicated in this way. Implementing P systems, whether in wetware or hardware, is still the subject of research, and at the moment P systems features are only useful from a theory of computation point of view [Păun and Rozenberg, 2002].

**Brane Calculi**

*Brane calculus* [Cardelli, 2005] stresses the importance of the conditions on both sides of the membrane by performing computation on the membrane rather than inside the membrane. Membranes are thus not here only containers but also take an active part in computations. The main underlying motivation behind this method is that biological membranes form a two-dimensional fluid embedded in a three-dimensional fluid, leading in this calculi to the existence of two monoids instead of one. This organisation creates a space where orientation must be conserved and thus membranes of different orientations should not be able to merge. It also involves bitonality which requires nested membranes to have opposite orientations. Structural congruence found in calculi like $\pi$-calculus is applied to membranes and to their content, the systems. The composition of systems uses the $\circ$ operator with unit $\circ$ while the composition of membranes uses the $|$ operator.
Chapter 2 Computational Biomimetism

with unit $\theta$. Actions can be defined to incorporate external material into membranes or to expel material from membranes.

Projective Brane Calculus [Danos and Pradalier, 2005] suggests a refinement of the brane calculus by replacing actions with directed actions. They argue that molecular implementations of actions are directed in biological membrane interactions, and therefore that enriching the brane calculus with this notion enables a more accurate description of biological membranes. [Danos and Pradalier, 2005] provides an example involving a big membrane, itself containing two smaller membranes. The two inner membranes respectively contain $P$ and $Q$. Two actions, $f_1$ and $f_2$, aiming at fusing membranes are provided. Actions are given, between the symbols $\langle$ and $\rangle$, just before the membrane they act on. The notation uses the semi-column symbol to separate inwards, to the left, from outwards, to the right, actions. Membranes are represented using the $L$ and $M$ symbols. The notation for the example is given below:

$$\langle ; f_1 \rangle L \langle f_1, f_2 ; \rangle L P, \langle f_2 ; \rangle L Q, MM$$

The two fusions $f_1$ and $f_2$ can happen in any order, whether mating $P$ and $Q$ before exocytosis (expelling material out of the membrane) or expelling both $P$ and $Q$ in turn. However both converge to the same solutions, as shown below.

$$\langle ; f_1 \rangle L \langle f_1, f_2 ; \rangle L P, \langle f_2 ; \rangle L Q \xrightarrow{f_2} P, \langle ; \rangle L Q, MM$$

Bio-graphs

Bio-graphs [Cardelli, 2003] is a graphical approach to calculus that is designed to represent biological systems using three graphical constructions: membranes, reagents and bindings. Membranes represent boundaries inside which reactions can occur. They can contain reagents and other membranes. Typically membranes interact with each other through the reagents, biological entities which usually represent proteins complexes. Reagents can also interact with each other and exchange binding sites which are pure...
names (i.e. have no further structure than their identifying name). Bindings can be represented by binding sites. These sites can be enclosed in binding boxes which define an abstract space (no physical existence, as opposed to membranes) where sites are privately shared (no other entity can see them from outside the box).

There is a fixed set of primitive reagents (8 reactions) defining a Turing complete formalism. Membrane reactions are for a membrane to enter, exit or merge with another membrane. Site reactions, which do not affect membranes, occur on binding sites and reagents can exchange tokens (binding sites or other entities) locally or across membranes. Finally another reaction enables replication of reagents.

Figure 2.21 provides these graphical notations along with an example given in [Cardelli, 2003]. The example provides a simple model of a symporter, which here binds two proteins from outside the cell and then brings them inside the cell. Figure 2.21(c) shows the symporter model and Figure 2.21(d) shows the setup and result of the symporter placed into a cell surrounded by proteins. The symporter binds in any order proteins P and Q and takes them inside the cell. The pure names p and q remain protected by the binding boxes which ensures that nothing else can interfere with the proteins any more. The repeat reagent (one of the 8 reactions) allows this process to repeat indefinitely. The cell and protein behaviour text in Figure 2.21(d) stands for the behaviour of cells.
and proteins that could be modelled in addition but is not needed for the symporter model.

The bio-graphs approach provides a better readability over pure algebra notations such as $\pi$-calculus. [Cardelli, 2003] states that they can also be seen as a special case of bi-graphs [Milner, 2001]. Bio-graphs were designed to model biological systems at a molecular scale like other calculi inspired by membranes. This focus on one particular approach of natural computation might be restrictive or inappropriate for other approaches at higher levels of abstraction. Also the concept of binding through pure names remains very similar to the concept of channel found in $\pi$-calculus which, as discussed above does not make clear what is linked to what. In the symporter model, the resulting state makes clear that the proteins are inside the cell, yet the binding process was not modelled beyond an exchange of pure names. By remaining anchored into traditional calculi approaches, similar design issues keep reoccurring. The next paradigm addresses these issues by also representing interactions, and not just their result.

Systemic Computation

According to Bunge in his “Treatise on basic philosophy”, every science studies some forms of systems, whether physical, chemical, biological, social or artificial [Bunge, 1979]. In order to provide a unified framework for the study of systems disregarding their origin, systemics was coined as a set of theories that focus on the structural characteristics of systems and can therefore cross the barriers between disciplines [Bunge, 1979]. Watson defines systemics as the root scientific discipline, underpinning other branches of science by studying the origin of organisation per se [Watson et al., 1999]. The notion of systemics is closely related to the fields of cybernetics and systems theory. Cybernetics [Wiener, 1948] is an interdisciplinary field that studies the functioning of regulatory systems. It is also a generic approach focussing on the structure and principles of systems involving circular causality. It has been largely inspired by pioneer works such as von Neumann’s universal constructor (discussed above) among others [François, 1999]. François states that systemics and cybernetics can be viewed as a metalanguage of concepts and models for transdisciplinarian use [François, 1999]. Systems theory is another interdisciplinary field that considers systems as a set of independent and interacting parts in order to study general principles of their functioning. The concept mainly refers to von Bertalanffy’s general system theory [von Bertalanffy, 1950] and is also closely related to cybernetics. Cybernetics can indeed be seen as a subset of systems theory focussing on systems with feedback loops. A systemic view encompasses all these notions but also stresses the importance of holism [Smuts, 1926]. A holistic analysis of systems considers them as being more than the sum of their parts. It hence acknowledges that understanding their separate parts individually is not sufficient to understand them as a whole. A
systemic approach is therefore a generic and holistic approach to systems where systems can adopt a recursive definition and be seen as a set of interacting sub-systems.

Systemic computation (SC) was introduced by Bentley as a novel computing paradigm and a suggestion of necessary features for a computer architecture compatible with current processors, memories, sensors yet designed to provide native support for common characteristics of biological processes [Bentley, 2007a]. As such it has been designed to support bio-inspired models and algorithms while sharing the desirable features of biological systems not found in conventional architectures [Bentley, 2007a].

Systemic computation presents the following aims:

- Support nature-inspired algorithms by using rules enabling accurate, elegant and efficient nature inspired models or algorithms.
- Provide a software engineering base resembling natural systems and thus addressing the issues of software brittleness found in conventional software.
- Provide a computer architecture reflecting the software organisation to enable a full and transparent hardware support for the software.
- Provide a methodology and tools enabling an accurate modelling and analysis of nature-inspired models.

To highlight the potential of systemic computation, Bentley initially provided a demonstration of Turing completeness, a first experimental platform to implement basic programs and a suggestion of nature-inspired models [Bentley, 2007a]. He later described a potential systemic computer based upon sensor networks [Bentley, 2007b].

Turing Completeness

As previously discussed, a Turing Machine [Turing, 1936] defines an abstract model of machines able to simulate the logic of any computer algorithm. Conventional computers belong to the family of Universal Turing Machines (UTMs). Such machines are able to simulate any other Turing machine, including other UTMs. A computational system is Turing complete if it is able to simulate any UTM. By definition of UTMs if such system can simulate a given UTM it can thus simulate any UTM. Proving that systemic computation is Turing complete is an important result as it implies that any algorithm, model, program that can run on a computer (or any other known Turing machine) can run on a systemic computer.
While UTMs all have the potential to simulate each other, the execution speed of the simulation of one by the other can be significantly lowered compared to a native hardware machine. As a result deterministic Turing machines (DTM) can simulate non-deterministic Turing machines (NTM) [Francez et al., 1978] and vice versa but the speed penalty of one simulating another often requires dedicated hardware to function properly, as discussed [Bentley, 2007a]. Providing SC as a Turing machine designed for nature-inspired systems would remove the limitations of conventional architectures for this domain of application.

In this respect, Bentley provided a Turing-completeness demonstration for SC by showing how to implement a cellular automaton with rule 110 within SC. A cellular automaton implementing rule 110 was indeed demonstrated to be Turing complete in [Cook, 2004]. SC being able to simulate this UTM, it can consequently simulate any UTM. SC is therefore theoretically Turing complete. However, in the demonstration [Bentley, 2007a] only provides a description of a potential implementation. A complete Turing-completeness proof based on such implementation would require a full description of the implementation on a given systemic computer and experimental results to in addition empirically validate the theoretical proof.

**First Experimental Platform**

As the motivations were not only to suggest a theoretical approach to the modelling of natural systems but also to produce a practical architecture for these models, an implementation of a systemic computer as a virtual architecture has also been presented in [Bentley, 2007a]. It defined a low level instructions set, a machine code and an assembly language with compiler.

While this implementation was a first proof of concept as a functional systemic computer, it only implemented thirty low level functions (e.g. basic mathematical or logic operators) and did not provide a framework for potential users to define more. This computer therefore only provided a limited range of potential interactions and was not suitable for higher level interactions (e.g. crossover of two solutions in a genetic algorithm). Also the language would only consider schemata and transformation function, allowing no static storage space for data within context systems (e.g. data specific to each context such as an identifier).

A more flexible platform with extended language features is therefore required to build high level models of any kind (e.g. crystal growth, genetic algorithm, artificial immune system). This is addressed later in Chapter 4.
Sensor Networks Implementation

Another platform approach is discussed in [Bentley, 2007b] to show how a physical implementation of this architecture could be achieved through the use of wireless devices produced for sensor networks. Beyond its usefulness for the understanding of natural systems, the aim of this work was to provide an architecture for a fault-tolerant and autonomous computer that could exploit all the features of sensor networks while providing benefits for robust wireless networking.

First Models Suggestions

SC being designed for the modelling of bio-inspired processes, [Bentley, 2007a] also suggested implementations of nature-inspired computer models or algorithms such as a cellular automaton, fuzzy logic, cell division, neural network and evolutionary algorithm within SC.

While these models seem appropriate for SC, no functional or conceptual implementation was suggested to demonstrate their feasibility. To address this and assess the appropriateness of SC for such models this thesis provides in further sections concrete implementations of a genetic algorithm, an artificial neural network, an artificial organism, a bistable gene network and a mitogen-activated protein kinase cascade.

Finally, Bentley also provided SC implementations of some basic conventional operations. Even though SC was not designed for this kind of operations, this illustrates how conventional computation could be performed in SC [Bentley, 2007a]. A similar example is detailed in Section 3.5.

Properties Analysis

In [Bentley, 2007a] natural computation is opposed to conventional computation with a list of computational properties respectively found in both types of computation. However no distinction was made between what this work defines as native properties and behavioural properties. As previously defined, behavioural properties are advanced properties enabled in computer models by the exploitation of some native properties (and/or potentially acquired behavioural properties).

This thesis also lists new properties such as the native property of local knowledge, found in natural computation, as opposed to global knowledge, found in conventional computation. Computation can indeed be asynchronous, parallel and distributed while sharing a common data access. Natural systems clearly work in a different way and function with the local information, like cells in an organism or ants in their environment. This concept was clearly acknowledged in the initial work [Bentley, 2007a] but not
formulated as a property. The behavioural property of self-organisation is also added in this thesis.

2.5.4 Summary

This section presented alternative paradigms of computation that have been used to approach natural processes. Other approaches have been developed but this section focussed on the most widely used or significant. Table 2.4 provides for the reviewed approaches an evaluation of the level of achievement of the natural computational properties, as well as an evaluation of the behavioural properties that the techniques are expected to achieve in some models.

Table 2.4: Evaluation of the natural properties achieved by the reviewed computational paradigms. The evaluation ranges from no dot, if the property is absent, to three dots if the property is fully supported. For computational paradigms some properties are dependent upon the models realised or the use of the paradigm within a specific hardware context. The empty dots used here therefore indicate an estimation of how well properties are theoretically or empirically achieved, based upon the aim of the method as suggested by authors or related work on the method. Note that SC is marked full as it was explicitly designed to provide the properties of natural computation; some properties are however theoretical and unproven at this stage.

<table>
<thead>
<tr>
<th>Property</th>
<th>Maths, physics &amp; technology inspired</th>
<th>Nature-inspired</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CGPs</td>
<td>π-calculus</td>
</tr>
<tr>
<td>Stochastic</td>
<td>•</td>
<td>**</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Parallel</td>
<td>**</td>
<td>***</td>
</tr>
<tr>
<td>Continuous</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Distributed</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>Approximate</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Embodied</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Circular causality</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Local knowledge</td>
<td>***</td>
<td>*</td>
</tr>
<tr>
<td>Compliance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Self-organised</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Homoeostatic</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Robust</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Autonomous</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Open-ended</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Complex</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Approaches based on concurrent computation, like the π-calculus, have not been designed to abstract the behaviour of natural systems and have a different approach to computation. Derived versions of the well explored π-calculus included new natural features but the origins of these paradigms tend to reflect mathematical concepts more than
natural computation. Also, most of these methods suggest a theoretical analysis of complex and nature-inspired systems. As discussed in previous sections, even if implemented in software as in [Phillips and Cardelli, 2004a], a model requires hardware backup to fully enable natural properties such as fault-tolerance. The difficulties encountered in hardware parallel implementations of P systems [Ciobanu and Wenyuan, 2004] suggest that a computation paradigm needs to also consider its potential implementation at its core. In this respect, distributed computational systems and new technologies might require a formalisation designed at its core for a digital yet natural computation providing the power and reliability of natural computation while maintaining full compatibility with modern hardware.

Because systemic computation has been introduced to address these issues it has the potential to demonstrate more natural properties more completely compared to other approaches (see Table 2.4). SC therefore presents an alternative approach where organisms and software programs share a common definition of computation that is designed to be implemented in hardware. However its potential has not yet been proven. It lacks concrete case studies along with modelling methodology to demonstrate the capabilities of SC computer models to show features such as fault-tolerance, homeostasis or self-organisation. It also lacks a platform allowing the implementation of such models. This thesis therefore focusses on SC in order to assess whether the potential natural properties can be achieved, and therefore assess the utility of this approach for the first time. The next chapter describes the methodology used for this work.
Chapter 3

Methodology

3.1 Introduction

The purpose of this work is to investigate how systemic computation enables exploitation and analysis of natural properties within nature-inspired computer models. This chapter describes the processes and methods developed and used to perform this investigation.

The focus of the work is on specific natural properties. The equipment that was chosen is a specific version of systemic computation that provides the necessary aspects for this study. To implement it, a complete platform (presented in detail in the next chapter) including specific language, compiler and virtual machine has been developed. Models can be expressed using a graphical and a calculus notation. They run following the systemic computing principle. Each model is developed using a systemic analysis. The testing of the natural properties was performed by implementing different natural models on the platform. The analysis of results was performed by making use of statistical analyses, and the use of graphical visualisation tools were also investigated.

3.2 Investigated Properties

Many properties found in natural systems could be of benefit for computer technologies if they were sustainable and obtained almost naturally. Systemic computation was designed to provide digital computation with the same capabilities as natural computation. This work will investigate the potential of SC for the properties of self-adaptation, fault-tolerance, crash-proof computing, self-repair, homoeostasis, flexibility and self-organisation in some nature-inspired models.
• Self-adaptation enables models to adapt to changing conditions. Such ability is an expression of robustness.

• Fault-tolerance provides a model with a tolerance to faults enabling a model to carry on working or to gracefully degrade.

• Crash-proof computing gives a computer the ability to carry on working in spite of any partial damage. This property is an expression of fault-tolerance at the computer level.

• Self-repair allows a fault-tolerant model to recover fully or partly from faults. It is therefore and improved feature of fault-tolerance.

• Homoeostasis gives a model the ability to remain stable by detecting changes in the model condition and acting on it to ensure the whole remains in a constant condition.

• Flexibility in some computer models that have not a definite structure can be an important part of the model. The way such program is built then allows the instantiation of various models with different structures, yet working with the same systems. Flexibility can also be found in models that can be changed or improved by adding systems to their existing hierarchy. Flexibility is another expression of robustness as modifying the structure or behaviour of a model would not require reprogramming its basic elements.

• Self-organisation in a program can be used to self-organise data depending on given criteria (e.g. similarities).

The properties hierarchy is summarised in Figure 3.1. Some of these properties can result from one or a combination of the natural behavioural properties listed in Subsection 2.2.5 (page 48). For instance, a system could not attempt to self-repair if it is not at least a little fault-tolerant beforehand.
3.3 Thesis Version of Systemic Computation

In [Bentley, 2007a], systemic computation was introduced with the set of general rules given in Section 1.2. These rules define the basic principles of modelling and interactions in SC. However, the work also defined more specific rules that defined scope range and schemata matching.

Scope defines the concept of embedded hierarchy. Systems can be contained by other systems. Interactions are constrained by scopes and therefore an interaction can only involve a context system and two interacting systems that are part of a same scope. Also systems can be in several scopes at once and as such interact within different scopes akin to different universes with their own rules but impacting one the other or one another through shared objects [Bentley, 2002]. In the work from Bentley, scopes were defined as infinitely recursive, fuzzy and overlapping.

- **Recursive scopes** allow systems to contain systems that contain systems and so on infinitely. The notion of containing self was however not defined. This thesis allows this notion and therefore a system can contain itself or a sub hierarchy containing itself. Such definition enables for instance the modelling of fractal sets such as the Mandelbrot set [Mandelbrot, 1980] where sub parts of a set can potentially be the set itself.

- **Fuzzy scopes** allow systems to partially belong to other systems. This concept allows to modulate the probabilities of interaction within scopes by changing the hierarchy and not the definitions of interactions themselves. Using this method, a system more part of a scope than another system has a higher probability to interact in this scope. This thesis focusing on the assessment of natural properties of computation in SC as opposed to conventional computation, the notion of fuzzy scope was not considered central (none of the models identified for investigation require such feature). Fuzzy scopes indeed allow a more precise hierarchy definition but does not provide an additional property. Its use would also add complexity to the implementation while not being necessary for the concepts or models developed in this thesis. In addition, whenever a probability of interaction is required, the description of the interaction can itself contain this probability (i.e. a probability \( p \) to perform or not the computation). This probability can also be set as an attribute variable (i.e. a data value) of an interacting system. This simplification has also speed benefits for the implementation presented in the next chapter, which will be discussed then. The fuzzy scope notion is therefore not used and implemented in this work.
• **Overlapping scopes** allow systems to be in the scope of one another. With fuzzy scopes, systems can be partly in each other’s scope. Such relation can be used to represent bound systems (e.g. two bound molecules). In the work of this thesis systems can only be fully in the scope of one another since fuzzy scopes are not enabled.

*Schemata matching* is the mechanism that allows a context system to identify the systems that are eligible for interaction in the given context. Schemata have a wildcard symbol that allows to define partial matching. Bentley also introduced the notion of *threshold matching* that allows a system to be picked even if it does not match some parts of a schema. The latter not being needed for the models implemented in this work, threshold matching was not implemented. This simplification also again has benefits for the implementation, as will be discussed in the next chapter. Similarly to fuzzy scopes, partial matching can nevertheless be simulated within the interaction function by performing a looser acceptance of systems for interaction (e.g. using wildcards to accept a wider range of systems) and apply in the interaction a probability to compute or skip the computation based on similarity with a given pattern.

Finally, all the systems defining a model can be wrapped (i.e. contained) in a super-system called *universe*. This convention guarantees a program hierarchy where all the program systems are contained in at least one system, thus without floating systems belonging to nowhere. This convention also provides a sensible interaction representation with the end-user in the context of a computer, as illustrated in Figure 3.2. This will be explored in more details in Chapter 4.

![Figure 3.2: Human-program interaction in the context of the systemic computer](image-url)

From a modelling perspective, the context of interaction of two systems should remain unchanged by this very interaction. In Bentley’s definition the context is never changed. In this work however, and in very specific cases, the context may be allowed to have a variable modified. Such situation should be seen as an approximation of a further interaction, involving one of the initial subjects and the initial context as a subject, that
alters this variable. For efficiency and simplification reasons, this approximation can be appropriate.

The version of systemic computation defined in this work therefore has the following rules:

- Everything is a system. (The system is the elementary unit of SC.)
- Systems can be transformed but never destroyed or created from nothing.
- Systems may comprise or share other nested systems.
- Systems interact, and interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems.
- Computation is transformation.
- A system is either fully in the scope of a system or not at all (out of the scope).
- Scopes are infinitely recursive and can contain themselves.
- A system is selected by a context for interaction only if the system’s definition fully matches the context’s schema.
- All systems in a model are contained directly or indirectly (through other systems) within a universe system.
- A context of interaction should not be changed during an interaction where it acts as context, unless exceptionally for a specific approximation purpose significantly benefiting the model or simulation.

### 3.4 Systemic Computation Notations

In order to describe models implemented using SC, a calculus notation as well as a graphical notation are provided. The calculus notation offers a textual but formal and precise way to describe models. The calculus notation has been used in [Le Martelot and Bentley, 2009a,b,c, Le Martelot et al., 2008b] to describe models such as an artificial organism, a chemical reaction and genes and proteins regulation networks. The
graphical notation provides an intuitive visual representation of the systems and their potential interactions. It is used to describe all SC models. In [Bentley, 2007a] a first graphical notation for systemic computation was presented but this work clarifies and helps formalise the graph and calculus notations in greater detail.

### 3.4.1 Calculus notation

The calculus notation used in this thesis is given in Table 3.1 and uses the following notations:

- Systems are represented by labels (i.e. names), as well as the attribute variables. (Attribute variables are where data can be held within a system, similarly to a variable in a data structure.).

- The schemata are represented using the textual notation } or {. However they are only shown when an interaction is occurring. (When a list of systems is given in a line, showing the schemata around a system could suggest that the system is context of interaction between its two neighbour systems, the one on the left and the one on the right, thus leading to an unclear notation. Therefore schemata are only shown when effectively describing an interaction.)

- The scope notation uses parenthesis.

- The transformation from one expression to another is denoted using the symbol $\rightarrow$. Note that for any interaction to be valid, the two interacting systems and the context system must share a common scope, not necessarily shown in a calculus expression when unnamed.

Note also that several notations are close but convey a specific additional information. 6 defines an interaction where the three systems (the two interacting systems and the context of interaction) are simply located within a common scope (which is a minimum requirement for interaction). 7 specifies that system$_2$ is nested within system$_1$. 8 indicates that system$_1$ and system$_2$ are in the scope of one another with context nested within system$_1$. 9 indicates that system$_1$ and system$_2$ are in the scope of one another with context nested within both. 10 says that system$_1$ and system$_2$ are in the scope of one another but with context located outside both of them. 11 shows an interaction with a parent system, where system$_2$ and context are located within the scope of system$_1$. 
<table>
<thead>
<tr>
<th>Expression</th>
<th>Signification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. system</td>
<td>A system called \textit{system}.</td>
</tr>
<tr>
<td>2. system[x_1, \cdots, x_n]</td>
<td>\textit{system} contains variables (x_1) to (x_n).</td>
</tr>
<tr>
<td>3. system(sub_1 \cdots sub_n)</td>
<td>\textit{system} contains subsystems (sub_1) to (sub_n).</td>
</tr>
<tr>
<td>4. (system_1 \cdots system_n)</td>
<td>Systems \textit{system}_1 to \textit{system}_n share a same unnamed scope.</td>
</tr>
<tr>
<td>5. (system_1 ( ) system_2)</td>
<td>Systems \textit{system}_1 and \textit{system}_2 are in the scope of one another (i.e. \textit{system}_1 contains \textit{system}_2 and \textit{system}_2 contains \textit{system}_1).</td>
</tr>
<tr>
<td>6. system_1 \rightarrow context {- { system_2 } }</td>
<td>Systems \textit{system}_1 and \textit{system}_2 interact in the context \textit{system context}.</td>
</tr>
<tr>
<td>7. system_1{system_2} } \rightarrow context</td>
<td>Systems \textit{system}_1 and \textit{system}_2, with \textit{system}_2 being also contained within \textit{system}_1, interact in the context \textit{system context}.</td>
</tr>
<tr>
<td>8. (system_1 \rightarrow context {- { ( ) system_2 } }</td>
<td>Overlapping systems \textit{system}_1 and \textit{system}_2 interact in the context \textit{context} within the scope of \textit{system}_1.</td>
</tr>
<tr>
<td>9. (system_1 ( ) context -{ ( ) system_2}</td>
<td>Overlapping systems \textit{system}_1 and \textit{system}_2 interact in the context \textit{context} nested within both \textit{system}_1 and \textit{system}_2. This notation is a contraction of 8 and its reciprocal (swapping \textit{system}_1 and \textit{system}_2), thus in a situation where the context is nested within the overlapping scopes. Interaction would happen only within one of the two merged cases at a time though (i.e. 8 or reciprocal).</td>
</tr>
<tr>
<td>10. (system_1 ( ) system_2) ) \rightarrow context</td>
<td>Overlapping systems \textit{system}_1 and \textit{system}_2 interact in the context \textit{system context} located out of both \textit{system}_1 and \textit{system}_2.</td>
</tr>
<tr>
<td>11. system_1 ( ) context -{ system_2}</td>
<td>Systems \textit{system}_1 and \textit{system}_2 interact in the context \textit{system context}, with \textit{system}_2 and \textit{context} being in the scope of \textit{system}_1.</td>
</tr>
<tr>
<td>12. S, S = system_1</td>
<td>\cdots</td>
</tr>
<tr>
<td>13. system[X], X = x_1 \cdots x_n</td>
<td>Generic name (X) that can take the various variable names (x_1) to (x_n). | here represents \textit{or}. This notation is used to merge several expressions where one variable could be replaced by another. (E.g. system_1[X] \rightarrow context {- { system_2, X = x_1 } ) \rightarrow context -{ system_2 } , X = x_1 } x_2 }</td>
</tr>
<tr>
<td>14. interaction → result</td>
<td>\textit{interaction} is a triplet comprising two interacting systems and a context of interaction (6-11). \textit{result} is the interaction result showing the organisation of the systems (3-5). In \textit{result} the context of interaction can be discarded assuming it remained unchanged. Both \textit{interaction} and \textit{result} expressions can include variable notations (2), as well as generic notations (12-13). \textit{result} can be several stochastic outcomes separated by (\mid). (E.g. s_1 \rightarrow inject {- { s_2 \rightarrow s_1 ( s_2 ) } ) \rightarrow context -{ system_2 } , X = s_1 } x_1 } x_2 }</td>
</tr>
</tbody>
</table>
3.4.2 Graphical notation

The graphical notation represents systems, hierarchies, interactions and their resulting scope changes and transformations. Figure 3.3 provides the graphical notations along with the corresponding calculus notations. A context system is displayed with its

![Diagram](image-url)

(a) Context system

(b) Non-context system

(c) Systems transformation

(d) System insertion

(e) System ejection

**Figure 3.3:** Graph notations for interactions and corresponding calculus notations. (a) shows a context system where schemata are drawn. (b) shows a non-context system, without schemata. (c) shows an interaction and its result where system\(_1\) and system\(_2\) got transformed into system\(_1'\) and system\(_2'\). (d) shows an interaction resulting in inserting system\(_2\) into system\(_1\). (e) shows an interaction resulting in ejecting system\(_2\) from system\(_1\).
schemata, not touching any other system if not involved in an interaction. (Note that as explained above the calculus expression does not show the schemata unless there is an interaction to describe.) A non-context system is displayed without schemata. (Note that any system can potentially act as a context, therefore a system not acting as a context in an interaction may act as a context in another one.) An interaction between two systems within a context where the systems are transformed can be indicated using dashed arrows going from the original system to the resulting system and passing by the context to indicate that the transformation results from an interaction in that context. (Note that when the two systems are transformed, depending on the nature of the transformation, the two dashed arrows could join and separate again in order to indicate that the two new systems result from a transformation of the two former systems as a whole, and not from a respective transformation of one former system into a new system.) The insertion and ejection of a system into and from a system is shown in a similar way. Note that in Figure 3.3(e) system₁ is not redrawn as it is actually not transformed, it just loses system₂. It was redrawn in Figure 3.3(d) for clarity as the scale had to be increased to fit system₂ inside.

For clarity, the graph notation does not have to show the variables or attributes systems may hold (i.e. data). These must be shown using the calculus notation. The concept of the graphical notation is to represent with colour and/or labels the various kinds of systems. Any major data that might affect the role of a given system should rather be represented using two neighbour colours or labels (e.g. assuming Figure 3.3(c) changes an attribute in system₁, it then becomes system₁’ instead of having textual data aside). Note also that the arrows notation is given to enable the drawing of both an interaction and its result within one drawing, but successive graph representations can also be given to illustrate the states of transformation.

Figure 3.4 provides graphical representations for the most common scope situations that can be encountered, along with the corresponding calculus notations. Note that the sharing of a system between scopes does not necessarily imply that the scopes are overlapping. However if considering neighbourhoods (e.g. in the real world as well as in a cellular automata) sharing systems, it can be sensible to also represent the scopes as overlapping (even if the model implementation is not computationally impacted by such additional information and does not need it coded).

The graphical representation of some scopes and systems sharing can sometimes be difficult to achieve in a fully non-ambiguous manner. Figures 3.4(b), 3.4(c) and 3.4(d) are close notations and systems shared by several systems but not the parent, or scopes overlapping and others not, can make the graphical notation tricky. Some relations might also be so complex that representing everything graphically becomes confusing.
In such case the graphical notations might then focus on some specific relations only, also using the calculus notation to specify what cannot be drawn.

(a) Overlapping scopes

(b) Shared system between overlapping scopes

(c) Shared system between two separate scopes

(d) Shared system between other scope and super scope

(e) Shared system between super and sub scopes

(f) Same system represented in two separate scopes

**Figure 3.4:** Graph notations for locality (potential scope situations and systems sharing) and corresponding calculus notations. (a) represents two overlapping scopes \( \text{system}_1 \) and \( \text{system}_2 \). (b) represents two overlapping scopes \( \text{system}_1 \) and \( \text{system}_2 \) sharing the system \( \text{system}_3 \). (c) represents two non-overlapping scopes \( \text{system}_1 \) and \( \text{system}_2 \) sharing the system \( \text{system}_3 \). (d) represents two non-overlapping scopes \( \text{system}_1 \) and \( \text{system}_2 \) sharing the system \( \text{system}_3 \) which also belongs to a common super system. (e) shows \( \text{system}_3 \) shared between \( \text{system}_1 \) and \( \text{system}_2 \) but with \( \text{system}_2 \) being in the scope of \( \text{system}_1 \). (f) represents \( \text{system}_1 \) twice at various locations (for clarity or to represent systems placed in more than two non-overlapping scopes), the two graphical instances being linked by a dashed red curve.
3.4.3 Comparison with the Stochastic \( \pi \)-calculus

To provide further clarification of SC as used in this thesis and to place it into context with other approaches, a comparison is now given with the stochastic \( \pi \)-calculus. As illustrated above, systemic computation can be expressed using a graph notation and a more formal calculus. Methods such as the stochastic \( \pi \)-calculus provide an alternative way of expressing natural processes formally. However, the origins of \( \pi \)-calculus lie in computer science and communications theory, which is evident from the non-intuitive method of expressing the interaction and transformation of entities. For example, the binding model \( H + Cl \Rightarrow HCl \) illustrated in Figure 3.5 and used in [Phillips and Cardelli, 2004b] to introduce and teach the stochastic \( \pi \)-calculus shows the difference in clarity of expression very clearly.

\[ \text{Figure 3.5: Molecular model of HCl (Hydrogen Chloride)} \]

In the stochastic \( \pi \)-calculus model, illustrated in Figure 3.6, \( H \) donates its electron which is shared between \( H \) and \( Cl \). \( Cl \) then loses an electron and \( H \) gains it to break the bond. Binding of \( H \) and \( Cl \) is never explicit as they are never actually linked in the model but simply transformed from \( H \) to \( H_{Bound} \) and \( Cl \) to \( Cl_{Bound} \). This confusion of abstractions (are just atoms being modelled or also electrons?) and confusion of bindings (which atom is bound to which?) results in a partial and somewhat cryptic model.

In contrast, when using SC the model can remain at one level of abstraction: the atom. There is no need to model individual electrons for such a simple model of binding. Instead, the scopes of each atom can overlap to reveal the binding of the atoms. The result in SC graph form given in Figure 3.7 looks like a familiar representation of the molecular model for \( HCl \) (see Figure 3.5), making the model remarkably intuitive and simple to understand. But most significantly, the systemic computation version includes the cause of the binding: the energy. The whole reaction between the two atoms is impossible without sufficient energy, a detail that is necessary to include in SC (the interaction between the atoms must occur in some context) but can be and has been ignored in the stochastic \( \pi \)-calculus model, resulting in an incomplete model.
Figure 3.6: Graphical representation and corresponding notation of the stochastic π-calculus binding model \( H + Cl \rightarrow HCl \) (from [Phillips and Cardelli, 2004b]). It is not obvious that \( H \) and \( Cl \) become bound to each other, despite this being the purpose of the model.

Both approaches enable multiple atoms and bindings to be modelled by the addition of parameters. However SC permits the modelling of this process at various levels of detail by revealing the deeper systems (protons, neutrons and electrons) hidden within the \( H \) and \( Cl \) systems without changing the overall structure of the model. By designing systemic computation explicitly for natural and biological computation, real advantages of clarity can be obtained compared to other modelling approaches.
3.5 Systemic Computing

A systemic computer (whether emulated or hardware) runs the building blocks of systemic computation: the systems. Compiled from the program, the systems carry out all computation according to the natural rules of SC (as specified above in Section 3.3).

An SC program differs subtly from conventional logic, procedural or object-oriented program both in its definition and in its goals. A procedural program contains a sequence of instructions to process whereas an SC program needs, by definition, to define and declare a list of agents (the systems) in an initial state, as illustrated in Figure 3.8. The program execution begins by creating these systems in their initial state and then continues by letting them interact indefinitely and stochastically. The outcome of the program is thus created from an emergent process rather than a deterministic predefined algorithm.

Nature-inspired processes like genetic algorithms, ant colony optimisation, artificial neural networks or artificial immune systems already partly operate based on this principle of interaction and emerging behaviour. However systemic computing forces their implementation to fully rely on this principle and bans the possibility to perform global computation without underlying local interactions. While imposing clear constraints on the model, these rules ensure a more faithful natural computation which is necessary to reproduce the features of interest found in natural systems.

Also, since SC is Turing complete, it is possible to perform conventional computation within SC, using basic operations. Note that SC was not designed to achieve such form of computation, hence performing for instance mathematical operations within SC does not resemble a mathematical statement like when using conventional computation.
(e.g. Matlab, C++). Yet, for the purpose of illustration, Bentley provided an example showing how to perform basic operations in SC [Bentley, 2007a]. A similar example to Bentley’s is given with its graphical notation in Figure 3.9 and then below using the calculus notations. The given notations describe the progression of a simple program performing the calculation: 
\[
\text{print}((a_1 - a_2) \times (a_3 - a_4))
\]

As can be seen on Figure 3.9(a), the systems respectively holding \(a_1\) and \(a_2\) are marked with a different colour. This stands for a difference in their declaration which permits the minus function to know what should be the left operand, and what should be the right operand, and therefore guarantees that the calculation \((a_1 - a_2)\) is performed correctly, instead of performing \((a_2 - a_1)\). Such implementation can for instance be performed using a variable with the possible values left or right. Also, the systems for each subtraction operation are nested in a specific scope, or computation space. This ensures that the operation \((a_1 - a_4)\) cannot happen. The subtraction operator and the eject system are shared within the two computation spaces to avoid duplicating them in each scope (thus saving resources). In the following stages on Figure 3.9(b), the program computes the subtraction operations, leaving the result in the left system, and the right system’s value.
set to null (nothing to compute). Then in Figure 3.9(c), the results are ejected from their respective scope (computation space) to be accessible for the multiplication. In Figure 3.9(d), the multiplication is performed and in Figure 3.9(e) the result is stored in one system while the other one’s value is set to null. Finally in Figure 3.9(f), the result is printed by interacting with the parent system (e.g. I/O system) which can be the universe, as described above in Section 3.3. The organisation of such program as well as the interactions occurring are described below using the calculus notation.

The initial program organisation (Figure 3.9(a)) is:

\[
\text{universe ( print mult space}_1 \text{ space}_2 )
\]

\[
\text{space}_1 ( \text{ op}_1[a_1,\text{left}] \text{ op}_2[a_2,\text{right}] \text{ eject minus })
\]

\[
\text{space}_2 ( \text{ op}_3[a_3,\text{left}] \text{ op}_4[a_4,\text{right}] \text{ eject minus })
\]

where \text{universe} is the systemic universe, \text{space}_1 and \text{space}_2 the computation spaces for the two subtraction operations, \text{print} the system with the print command, \text{mult} the system performing multiplications, \text{op}_i the operand systems holding values, \text{minus} the system performing subtractions, \text{eject} the system ejecting operand systems from their current computation space.

The first interactions to happen are the two subtraction operations (Figure 3.9(b)) taking place respectively in the two computation spaces (the scopes are not indicated here in the calculus expressions of the interactions):

\[
\text{op}_1[a_1,\text{left}] \rightarrow \text{ minus } \text{ op}_2[a_2,\text{right}] \rightarrow \text{ op}_1[a_1-a_2,\text{left}] \text{ op}_2[\text{null},\text{right}]
\]

\[
\text{op}_3[a_3,\text{left}] \rightarrow \text{ minus } \text{ op}_4[a_4,\text{right}] \rightarrow \text{ op}_3[a_3-a_4,\text{left}] \text{ op}_4[\text{null},\text{right}]
\]

then the calculated results are ejected from their respective computation space (Figure 3.9(c)):

\[
\text{space}_1 ( \{} \rightarrow \text{ eject } \{ \text{ op}_1[a_1-a_2] \})
\]

\[
\text{space}_2 ( \{} \rightarrow \text{ eject } \{ \text{ op}_3[a_3-a_4] \})
\]

then the multiplication can take place (Figures 3.9(d) and 3.9(e)):

\[
\text{op}_1[a_1-a_2] \rightarrow \text{ mult } \text{ op}_3[a_3-a_4] \rightarrow \text{ op}_1[(a_1-a_2) \times (a_3-a_4)] \text{ op}_3[\text{null}]
\]

and finally the result can be printed (Figure 3.9(f)):

\[
\text{op}_1[(a_1-a_2) \times (a_3-a_4)] \rightarrow \text{ print } \{ \text{ universe } \rightarrow \text{ op}_1[\text{null}] \text{ universe}[(a_1-a_2) \times (a_3-a_4)]
\]

Note that the subtraction and eject operations occur independently in separate and distinct scopes (space_1 and space_2). They can therefore occur in parallel.
3.6 Systemic Analysis

Systemic computation is an alternative model of computation that was designed to improve fidelity and clarity in the modelling of natural processes. This work is investigating natural properties within nature-inspired computer programs using the SC paradigm. It was therefore necessary to develop models that resemble their original natural process enough in order to display the properties of interest. To address this, a method of analysis, the systemic analysis [Le Martelot et al., 2007a], was developed.

Before any new program can be written, it is necessary to perform this systemic analysis in order to identify and interpret appropriate systems and their organisation. The systemic analysis provides a method for analysing and expressing a given problem or natural system more formally in SC. When performed carefully, such analysis can itself be revealing about the nature of a problem being tackled and the corresponding solution. A systemic analysis is thus the method by which any natural or artificial process is expressed in the language of systemic computation. The steps are in order:

1. Identify the systems: determine the level of abstraction to be used, by identifying which entities will be explicitly modelled as individual systems;
2. Analysis of interactions: determine which system interacts with which other system in which context system;
3. Analysis of structure: determine the order and structure (scopes) of the emergent program (which systems are inside which other systems) and the values stored within systems.

As an illustration a model of a genetic algorithm (GA) [Holland, 1975] designed for SC is proposed. The first stage is to identify the systems. The use of a GA implies the need for a population of solutions, hence a collection of systems, with each system corresponding to one solution, seems appropriate. (A lower-level abstraction might use one system for every gene within each solution, but for the purposes of this investigation, this would add unnecessary complexity.) The identification of appropriate low-level systems is aided by an analysis of interactions. In a GA, solutions interact in two ways: they compete for selection as parents, and once chosen as parents, pairs produce new offspring. The use of contextual systems (which determine the effects of solution interaction) for the genetic operations therefore seems highly appropriate, as shown in Figure 3.10.

Any selection and evolution methods may be used but at this stage there is no need to define the implementation of the solutions and the operator. (An implementation will be presented later in Chapter 5.) For simplicity here the selection and reproduction
Figure 3.10: A genetic operator acts as a context for two interacting solutions

are assumed to be happening at the same time with the pair of interacting solutions. (All conventional operators could be implemented, e.g. using selection systems to move solutions inside a gene pool system, from which other reproduction systems would control the interaction of parents to make offspring. Such complexity is unnecessary here, however.)

Once the systems and interactions are understood, it is necessary to determine the order and structure of the emergent program. For this, scopes (which systems are inside which other systems) and the values stored within systems need to be determined. In a GA, the population is usually initialised with random values, before any other kind of interaction can take place. This implies a two-stage computation: first all solutions must be initialised, then they are permitted to interact and evolve. One way to achieve this is to use a super-system as a computation space, and an initialiser system. If all solutions begin outside the computation space, then the initialiser acts as context for interactions between the empty solutions and the computation space, resulting in initialised solutions being pushed inside the space and then ready for evolution, as shown in Figure 3.11.

Figure 3.11: (a) The initialiser acts as context for interactions between non-initialised solutions and a computation space. (b) The result of the interaction, as defined by the initialiser, is an initialised solution pushed inside a computation space where it can then interact with other solutions in the context of genetic operators (not shown here).
The GA model structure can thus be represented as in Figure 3.12. All systems are contained directly or indirectly within the universe. Initially the non-initialised solutions are placed within the universe, along with initialiser contexts and computation spaces (only one is shown on the figure but more could be considered in order to have parallel pools of evolution, also potentially sharing solutions). Once the model is running, the initialisers can initialise the solutions and push them inside computation spaces where genetic operators can process and evolve solutions two by two.

3.7 Investigated Models

Now the equipment for this thesis has been introduced, in order to investigate the properties listed in Section 3.2 models are chosen depending on their relevance to one or several properties of interest and their suitability to be studied. The selected models are a genetic algorithm, an artificial neural network model and an artificial organism. The reasons for choosing them are discussed below:

- A genetic algorithm is a process progressing in an unknown environment that can even potentially change over time depending on the application. In that respect the ability to adapt is appropriate and a GA could benefit from an additional mechanism adapting its operators depending on their ability to provide convergence. A GA is therefore a suitable model for self-adaptation.
A genetic algorithm is also a partly distributed natural architecture. To investigate the potential of fault-tolerance within SC, the insight that is followed is the potential of redundancy. To investigate fault-tolerance a process with redundant components is thus suitable and the GA is selected again for this investigation. The crash-proof ability of systemic computing will also be assessed within this investigation. In addition, the self-repair ability will be assessed in this model, using the components redundancy to repair damaged parts. This ability also enables a homoeostatic behaviour. Modifications brought to the GA model by adding systems in order to add features also illustrate model flexibility.

- **Artificial neural networks**, by their distributed and massively connected architecture, can be used to create diverse neural structures. By developing a neuron model that can be connected with no or little restriction the ability to cope with various structures can be studied. A model of artificial neural network using a modified back-propagation algorithm and where the systems can be arranged in various manners will be developed to assess the structure flexibility.

- A new kind of program, an **artificial organism**, will be developed to study homoeostasis by providing the model with its own metabolism. The idea is to feed the organism with a flow of data and to grow and maintain the organism using this flow of data. The data is turned into cells and the diet of the organism thus processes the data. A metabolism should provide an autonomous process of detection of anomalies. When a datum is considered as an anomaly it should be separated from the regular data. The work therefore assesses the potential for an artificial organism to detect anomalies in its diet.

In addition, as the organism uses the data flow to grow, this model is as such also suitable in order to study self-organisation. The organism grows by self-organising its cells upon the data carried within the food. Cells group based on similarities, organs may appear, thus offering an on-line classification of the data. Also, visualising the inside of such organism will provide visual representations of the given data sets.

Two additional models have been later developed to investigate the use of on-line visualisation for the analysis of natural properties and behaviour of SC models:

- A **bistable gene network** (BGN) that presents an understood behaviour nevertheless unpredictable from the beginning is suitable for investigating the various shapes a model visualisation can take over time depending on the model behaviour. Each run of such model can start in a same initial state but randomly end in one of two
states. The aim of a visualisation is to provide the tools for a visual analysis of the shape of a model that should reveal its current state and history leading to this state. This model will be analysed over time to show how it grows towards one state or the other, and thus visually towards one shape or the other.

- A mitogen-activated protein kinase (MAPK) cascade that presents a large network with a known yet non-obvious behaviour will be developed in order to show how visualising such model enables an understanding of the role of each component leading to a thorough understanding of the global outcome.

The visualisation of such models aims at enabling an intuitive behavioural analysis of complex system through an information flow and organisational changes analysis.

Table 3.2 summarises the models investigated in this work along with the analysis they will be used for.

<table>
<thead>
<tr>
<th>Models</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic Algorithm</td>
<td>Self-adaptation</td>
</tr>
<tr>
<td></td>
<td>Fault-tolerance</td>
</tr>
<tr>
<td></td>
<td>Crash-proof</td>
</tr>
<tr>
<td></td>
<td>Self-repair</td>
</tr>
<tr>
<td></td>
<td>Features flexibility</td>
</tr>
<tr>
<td></td>
<td>Homoeostasis</td>
</tr>
<tr>
<td>Neural Network</td>
<td>Structure flexibility</td>
</tr>
<tr>
<td>Artificial Organism</td>
<td>Self-organisation</td>
</tr>
<tr>
<td></td>
<td>Homoeostasis</td>
</tr>
<tr>
<td>Bistable Gene Network</td>
<td>Information flow</td>
</tr>
<tr>
<td>MAPK Cascade</td>
<td>Organisational changes</td>
</tr>
</tbody>
</table>

### 3.8 Analysis and Assessments

In order to assess the models and the achievement of natural properties the models are run on the platform developed for this work. As discussed above the SC platform provides systemic computing rather than conventional computing. The systemic computing involves the natural computational properties discussed in Subsection 2.2.5 (page 48).

The evidence is gathered by running series of experiments. Experiments will consist of runs of models, each experiment using a different setup in order to analyse the correlation...
between setup and behaviour. A change in setup can be a parameter change or a rate change but also new systems addition, systems removal or quantity modification when systems are redundant.

There are three types of evidence:

1. An improved naturalness within SC compared to conventional approaches. Naturalness is defined here as the likeness to the concept being modelled. As this work is concerned with nature-inspired models, an improved naturalness means a closer resemblance of structure and behaviour with respect to nature, when compared to other modelling approaches. For example, in SC everything is a system and systems cannot be created from nothing or destroyed. There is therefore a form of energy conservation inherent to SC. SC also imposes scopes to mirror environments (e.g. fields, skins, boundaries), and contexts to specify contextual interactions and thus stressing the fact that two same entities can interact in different ways depending on the context. The aforementioned example of the $HCl$ molecule shows that SC forces the model to be more accurate than the stochastic $\pi$-calculus one by incorporating the cause of transformations within the model as context of interaction and allowing the reacting components to be organised in a way that reflects their relationship (i.e. bound together as opposed to remaining separated and renamed). The unnatural rules of $\pi$-calculus on the other hand lead to keeping the $H$ and $Cl$ atoms as separate bodies that are renamed depending on their binding. Also the use of channels is unclear from a modelling perspective.

2. The conciseness, compactness and readability of a model design or implementation compared to conventional models of the same thing demonstrates the native contribution of SC to the core of the model. For instance the stochastic and distributed computation in a genetic algorithm need not be simulated like in a conventional approach, resulting in a simpler implementation.

3. The ability of a model to exploit some native computational properties (and potentially achieved behavioural properties) to build up behavioural properties to a greater extent or more usefully compared to, or not achievable by other approaches. For example computational properties like stochastic, distributed and local computation can be exploited to build up a fault-tolerant model. The same model with such computational and now behavioural properties can be further exploited to obtain more behavioural properties like the self-repair ability. The brittleness of operating systems potentially crashing after a memory fault or the design of conventional computers not coping well with a hardware failure make them unsuitable for fault-tolerance and self-repair abilities whereas the alternative design of SC offers the possibility to enable such properties.
In order to analyse the behaviour of a model and assess its properties, data are collected during experiments as textual logs or visualisations. Logged data are collected over several runs in order to perform statistical analysis and plotting and assess the tendencies of models to achieve a particular behaviour. Mathematical modelling and analysis are also performed if needed in order to explain or predict the behaviour of models.

The use of graphical visualisation tools is also investigated to provide the end-user with an on-line visual output of the ongoing interactions and transformations happening within and undergone by the model. The SC visualisation tools are designed to improve the ability to analyse the natural properties and behaviour of models. The SC visualisation framework provides:

- Easier access to certain kinds of information compared to other approaches (e.g. visualise computation instead of reading and deciphering it from text);
- New information not previously available using other approaches (e.g. on-line visualisation of all components - the systems - and an overview of their computational past).

Note that when conducting experiments, due to the simulation of systemic computing on a conventional computer (see Chapter 4) runs are long to perform. As a result the amount of runs performed for each experiment is limited to a number sufficient to ensure consistency with little variation in the results.

### 3.9 Summary

This chapter presented the approach to the research conducted in this thesis. The properties that are investigated were identified, the version of systemic computation and the notations that will be used were presented. The specific version of systemic computation contains the necessary features for this work while being optimised for a virtual machine. Calculus and graphical notations provide intuitive and formal ways to describe and define models. To help clarify SC and place it into context a comparison between modelling nature-inspired systems using the stochastic π-calculus and SC was performed and showed that SC provides a clearer and more faithful way of modelling. The concept of systemic computing was then discussed and the systemic analysis which lays down the basis for the modelling in SC was introduced. The selected models for these properties investigations were discussed and the testing methods presented. The next chapter presents the platform developed for this work.
Chapter 4

Platform

4.1 Introduction

Systemic computation is a novel paradigm designed to support models and simulations of any kind of nature-inspired system but SC also suggests a new computer architecture where software models run natively on a dedicated machine without any operating system interfacing. Before a hardware systemic computer could be considered, and in order to investigate and assess the potential of SC, building a virtual computer as a proof of concept is appropriate. The work in this thesis will focus on the exploitation of such virtual computer. This chapter presents the platform developed in order to perform such tasks. This platform models a complete systemic computer as a virtual machine within a conventional computer. It involves a programming language, a compiler, a virtual machine and a runtime environment. This work is therefore as such also a suggestion regarding how SC can be implemented. This chapter presents how the platform implements SC. Full details on how to use the platform runtime environment are given in Appendix A.

4.2 Virtual Machine

In order to perform systemic computing, a virtual machine written in C++ and based on portable code has been developed for conventional computers. Due to the very nature of conventional computer, such simulation differs from a real hardware computer in its implementation of parallelism, as explained below. Yet, while the virtual machine is slower than a potential hardware machine and not using native (i.e. hardware) parallelism, it nevertheless enables to run and study SC models for the analysis of SC properties they exploit or exhibit.
4.2.1 Systemic Computing Simulation

Since an SC program runs indefinitely, the virtual machine (VM) has to run an infinite loop. SC is based on parallel, asynchronous and independent systems; therefore the VM can simulate this by randomly picking a context system at each iteration. Once a context is selected, eligible subject (interacting) systems located within the same scope(s) are identified. If any are found, two of them are randomly chosen. A subject is eligible if its definition matches the schema of a context. The VM then executes the context interaction instructions to transform the interacting systems and thus process a computation. (Interaction instructions are discussed later in the chapter.)

A context may be isolated from its potential subjects, so a computation may not occur at each iteration. It may also not occur if the computing function had nothing to compute from the context and systems it was provided with (e.g. probability of computing or not leading to skip the computation).

In a real parallel machine, several if not all contexts can potentially be carrying computations at the same time. By making the approximation that computation time is similar for all interactions, all interactions that can happen in parallel would thus occur the same amount of time over time. The virtual machine, working sequentially at its core, can simulate this behaviour by holding a list of context systems from where to pick the next context of interaction, and then removing it from the list. When this list is empty, it is reloaded with all the context systems present in the model. It ensures a fair distribution of computation time between systems. All the computations performed by the contexts of the list until the list is empty will be called a cycle. A new cycle thus starts each time the list is reloaded.

When a new context system is created from an interaction, this new context is not added to the current context systems list. It is simply one more context system present in the model that will be loaded in the list with the others upon the next cycle. However if a context system is turned into a non-context system by an interaction, the ex-context system is removed from the context list.

Algorithm 4.1 gives the algorithm used by the virtual machine’s engine to simulate systemic computing. In the VM, random number generation is performed by a pseudo-random number generation seeded using system time.

Note that making the approximation that computation time is similar for all interactions may slightly change some probabilities of interaction. The potential computation asynchrony resulting on a parallel machine from the different time each independent process needs to execute is not reproduced. Therefore an interaction requiring twice more time
Algorithm 4.1 Virtual machine’s engine algorithm

1: repeat
2:   if context_list is empty then
3:     Load context_list with a reference to each context system in the model
4:   end if
5: context := Randomly pick a context from context_list
6: Remove context from context_list
7: context_scopes := Create a list of references to all scopes of context
8: repeat
9:   Pick up scope from context_scopes and remove scope from context_scopes
10:  Clear list eligible\textsubscript{1} of eligible systems for the 1\textsuperscript{st} schema
11:  Clear list eligible\textsubscript{2} of eligible systems for the 2\textsuperscript{nd} schema
12:  if scope matches schema 1 of context then
13:    Add scope to eligible\textsubscript{1}
14:  end if
15:  if scope matches schema 2 of context then
16:    Add scope to eligible\textsubscript{2}
17:  end if
18: for all systems system (except current context) within scope do
19:   if system matches schema 1 of context then
20:     Add system to eligible\textsubscript{1}
21:   end if
22:   if system matches schema 2 of context then
23:     Add system to eligible\textsubscript{2}
24:   end if
25: end for
26: until a pair of interacting systems can be made from eligible\textsubscript{1} and eligible\textsubscript{2}
27: system\textsubscript{1} := Randomly pick a system from eligible\textsubscript{1}
28: system\textsubscript{2} := Randomly pick a (different) system from eligible\textsubscript{2}
29: Apply interaction between system\textsubscript{1} and system\textsubscript{2} in the context of context
30: if system\textsubscript{1} was and no longer is a context then
31:    Remove system\textsubscript{1} from context_list if present
32: end if
33: if system\textsubscript{2} was and no longer is a context then
34:    Remove system\textsubscript{2} from context_list if present
35: end if
36: until end of simulation

than another one on a parallel machine would be executed once while the other would be executed twice. By ignoring these durations on a serial implementation, each of the two interactions would happen the same amount of times, thus affecting overall their probability to occur.

The rule of system picking is not exactly the same as in the implementation of [Bentley, 2007a] where systems where picked randomly (also using a pseudo-random number generator) provided at least three systems were present in a scope and the context system
had indeed an interaction function. Therefore no context list was managed in this implementation. Also any system changed by an interaction would benefit from a higher probability to be picked from the next interaction onwards. This bias gives priority to recently modified systems, making the assumption that changed systems are more likely to initiate a cascade of effects in the model. Contrarily, the implementation of this thesis gives priority to contexts that have not interacted most recently, ensuring that no context of interaction would be less likely to occur just due to an independent event resulting in other systems having changed more recently.

In Section 3.3 some simplifications were made, compared to the full initial SC description from [Bentley, 2007a]. These simplifications ruled out the partial schemata matching and the fuzzy scope principles. However, as the virtual machine is simulating systemic computing which is a computationally heavy task, optimisations of the engine algorithm to gain speed at simulation have a significant impact. In this respect, the simplifications made in Section 3.3 have some benefits for the virtual machine runtime speed:

- Fuzzy scopes would change the rule of selection of interacting systems (lines 27 and 28 from Algorithm 4.1) by giving each system a different probability to be picked depending on its belonging to the scope (e.g. using a roulette-wheel algorithm). This would add significant overhead at each iteration in the simulation process. However, as previously discussed, the interaction function can involve a probability to compute or skip its turn, therefore enabling the use and handling of probabilities only when needed, without a global mechanism slowing down the simulation.

- Similarly, partial schemata matching would step in at the creation of eligible systems lists (lines 12 to 24 from Algorithm 4.1). A system could be eligible not only if it matches the corresponding schema but also if it matches it enough (with a threshold providing a limit of acceptance). This process would slow down the algorithm by requiring a counting of the differences between each considered system and the schemata at each iteration in the simulation process. This process of partial matching can however also be simulated within the interaction function if needed, as previously discussed.

### 4.2.2 Program Loading and Memory Management

The implementation proposed in [Bentley, 2007a] provided partial schemata matching and fuzzy scopes but only provided a fixed amount of interaction functions (30), with fixed string length (for the schemata and the kernel) and fixed alphabet. The implementation presented here is fully generic and enables a user to define any interaction
function, to use systems of any length and potentially choose its encoding alphabet. This allows full flexibility to program and investigate various models involving basic or more advanced local interactions. This platform is therefore a much higher level platform, fully programmable.

In order to program the machine, a descriptive programming language was provided with a compiler, as defined in the next section. This provides the VM with a compiled description of the systems, using interaction function calls. However, considering the infinite diversity of potential interaction functions, the function themselves are coded separately in C++ plugins in order to benefit from the full language diversity of expression (especially useful for coding non-trivial functions). Therefore, to execute an SC program, the VM first loads a program description compiled from an SC program. This program contains references to functions that can be found in associated plugins, written in C++. These functions are compiled in one or several plugins for the VM as dynamic libraries. The VM dynamically loads these associated plugins when initialising a new model, and for each interaction that happens in the model at runtime the VM looks for the function implementation within the loaded plugin(s). All these dynamic linkages are slower to handle compared to a static set of functions but offer an infinite diversity of potential interactions. The full flexibility therefore came at the cost of a slightly slower execution speed.

In order to remain as close as possible to a real parallel implementation, the scopes of each system are held locally within these systems. Therefore each system keeps track of its super and sub systems. The system memory is illustrated in Figure 4.1. The systems

![System memory](image)

*Figure 4.1: System memory: the size of the schemata and kernel is fixed but the size of the scope lists can vary*

are held in a memory array initialised at program loading. Each system’s string has a fixed size for the duration of the simulation as system’s length is defined per model. However the memory array is not guaranteed to remain of fixed size due to the additional amount of memory the scope lists might require when systems join the scope of other systems (i.e. more super or sub systems). (Note that a real hardware platform would have a fixed amount of physical memory; the scope lists would therefore have a limited length, the memory available for each system being shared between schemata, kernel and scope lists.) Yet, the systems array and the scope lists are implemented using the STL C++ library *vector* class that keeps a memory capacity equal or greater than the
actual size of the vector, reallocating space only when this capacity is exhausted. This reallocation process was designed to only happen in logarithmic frequency with respect to the vector’s size \(^1\). As a result, dynamic memory allocation might happen several times during runtime, slowing the speed down compared to an implementation using a static array. However the speed loss occurrence decreases as they happen. The implementation from [Bentley, 2007a] used a global scope table to keep track of the scopes of each systems. This enabled to have a fixed size memory array for the systems, their length being fixed, and the scopes data being held elsewhere. This implementation relying on a global scope table is thus not adapted to a fully decentralised implementation. It also does not accept new resources. While systems cannot be created from nothing some models could consider exchanging systems with the real world (the real world being then part of such model). Therefore if new resources were to be introduced in the systemic computer, because it keeps the scopes information locally the implementation of this thesis could make use of them while a static scope table would not have this flexibility. Such models are however not investigated in this thesis.

### 4.3 SC Programming

To enable the creation of effective programs for a systemic computing platform, a specific descriptive SC language has been created to enable systems definition, and instantiation of systems within a hierarchy. The definition of interactions is then done separately in a C++ file.

#### 4.3.1 SC Language

The SC language is a descriptive language intuitively very close to the SC model that has been created together with a compiler translating source code into byte-code for the virtual machine. The aim of the SC language is thus to aid the programmer when defining systems, declaring instances of them and setting scopes between them. A first language had been defined in [Bentley, 2007a], the language presented in this work is partly inspired by this first language but extends significantly its features by allowing any form of interaction, more flexibility in systems organisation (e.g. length, context function’s location, broader use of the kernel memory, choice of the compression code), advanced language expressions (e.g. compression operator, \textit{wildcard or} operator, temporary variables for digital values, arithmetic expressions, array notation) and cleaner syntax to make programming more powerful and easier. This section describes the language and the potential errors that can occur at compilation.

\(^1\)STL vector reference: \url{http://www.cplusplus.com/reference/stl/vector/}
Chapter 4 Platform

Systems Definition and Code Compression

Defining a system involves defining its kernel and its two schemata. A system is then defined as a triple string: the first schema, the kernel, the second schema.

When a system is not acting as a context, its definition potentially contains 0, 1 and the wildcard ?. When a system acts as a context, the two schemata are used as the templates of the two interacting systems, and the kernel encodes the nature of the interaction. This raises the problem of coding a schema knowing that it has to specify complete systems (defined by a kernel and two schemata). The method chosen was to compress information making up each schema [Bentley, 2007a]. A compression code is used for this purpose, coding three bits (where each bit may be 1, 0 or the wildcard ?) into one character. This allows the complete description of a system (kernel and two schemata) in one single schema.

The compression of the template into a coded string does not have to be left to the user. A compression operator specified by using the square brackets [ ] is provided to allow the user to write a template like a system definition, without bothering with the compression which will be done automatically at compilation. Bits to be compressed are therefore placed within the compression operator [ ]. Table 4.1 gives the compression code used in this thesis (and close to the one used in [Bentley, 2007a]).

The code can however be chosen by the user if another compression code is preferable. The only important feature to respect in order to provide a valid code is to have the 27 possible combinations listed and for each of them a corresponding distinct ascii character being neither 0, 1, the wildcard ? or any special character used in the SC language (e.g. brackets, coma). These characters are indeed already used for other purposes. (Note that in this respect, the code from [Bentley, 2007a] would not be valid here as using 0 and 1 as code characters.)

<table>
<thead>
<tr>
<th>code</th>
<th>value</th>
<th>code</th>
<th>value</th>
<th>code</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>000</td>
<td>i</td>
<td>100</td>
<td>r</td>
<td>?00</td>
</tr>
<tr>
<td>a</td>
<td>001</td>
<td>j</td>
<td>101</td>
<td>s</td>
<td>?01</td>
</tr>
<tr>
<td>b</td>
<td>00?</td>
<td>k</td>
<td>10?</td>
<td>t</td>
<td>?0?</td>
</tr>
<tr>
<td>c</td>
<td>010</td>
<td>l</td>
<td>110</td>
<td>u</td>
<td>?10</td>
</tr>
<tr>
<td>d</td>
<td>011</td>
<td>m</td>
<td>111</td>
<td>v</td>
<td>?11</td>
</tr>
<tr>
<td>e</td>
<td>01?</td>
<td>n</td>
<td>11?</td>
<td>w</td>
<td>?1?</td>
</tr>
<tr>
<td>f</td>
<td>0?0</td>
<td>o</td>
<td>1?0</td>
<td>x</td>
<td>??0</td>
</tr>
<tr>
<td>g</td>
<td>0?1</td>
<td>p</td>
<td>1?1</td>
<td>y</td>
<td>??1</td>
</tr>
<tr>
<td>h</td>
<td>0??</td>
<td>q</td>
<td>1??</td>
<td>z</td>
<td>???</td>
</tr>
</tbody>
</table>

To illustrate this, a system with definition 010 111 ??? would thus be compressed as cmz. A context system that needs to match such system in a schema would then have
this compressed string in the schema. This example assumed that the system defined in
the schema is made of binary or wildcard values. If a context system needs to define in
a schema an expression that matches another context system (e.g. cmz 111 cde) then
the problem of compressing compressed expressions arises. However, the schemata of a
context system provide information about the systems to reach to create an interaction,
but not about the context system itself. Information regarding the context are held in
the kernel part of a system which is never compressed (i.e. always made of binary or
wildcard values). Therefore describing a context system in a schema should be done by
describing its kernel. (Later in the chapter a method of system typing is presented to
help define and identify systems, whether context systems or not.) The context system
cmz 111 cde would therefore be defined in a schema as ??? 111 ?? where 111 is its
kernel and therefore compressed as zmz.

Labels

In order to make the code more readable, string labels can be defined and then used in the
program instead of their value. Because all systems can potentially be interchangeable
(any transformation can potentially happen) they need to be the same length. So that
compression code and schemata matching works, the two schemata and the kernel also
need to be the same length. The length of all schemata or kernel in a program is called
the word length. All labels must have the correct word length. Since the length bounds
the amount of information a system can store, the choice of the length is left to the user.
Labels are defined using the keyword label, as shown in Listing 4.1.

To combine labels, a wildcard or operator specified by | is used in combination with
the wildcard symbol. Labels defining different bits within the same schema can then
be combined using the operator |. Table 4.2 illustrates the operator’s use: Note that

\[
\begin{array}{c|c}
\text{expression} & \text{value} \\
\hline
0 | ? & 0 \\
1 | ? & 1 \\
? | 0 & 0 \\
? | 1 & 1 \\
? | ? & ? \\
0 | 0 & \text{illegal operation} \\
0 | 1 & \text{illegal operation} \\
1 | 0 & \text{illegal operation} \\
1 | 1 & \text{illegal operation} \\
\end{array}
\]

operator | is not a binary or since an operation such as 0 | 1 is not allowed.

For instance if \( \text{LABEL}_1 \) and \( \text{LABEL}_2 \) are respectively set to 10?? and ??01 then the
expression \( \text{LABEL}_1 \mid \text{LABEL}_2 \) means 1001.
**Interaction function**

The kernel of each system defines the function to be applied to matching interacting systems (and can also store variables). These functions potentially transform the interacting systems’ values and scopes. A function label is defined to create a mapping between a binary value held in the kernel and a function name. A function name must refer to a function implemented within an associated plugin. Such labels are defined using the keyword *function*, as shown in Listings 4.1 and 4.2.

Function labels can be combined with other labels using the *wildcard or operator* |. Function labels indeed only define the function code in the kernel but the kernel might hold additional data. For instance function label 11????, where the first two bits define the function, and label ????01 could be combined with | to give 11????.

The *NOP* function name stands for *nil function* and therefore does not refer to any function implementation. A lookup table matches the kernel values encoding function names with the corresponding call-back functions. This range of values is later defined in the encoding section.

**Basic system definition**

To illustrate the SC language, Listing 4.1 provides code defining a simple non-context system, here called *MySystem*.

**Listing 4.1:** Definition of a non-context system using a word length of 4 and the first two characters of the kernel as function code

```
label MY_SYSTEM 0001
label MY_K_DATA ??01
label MY_S_DATA 1010

function NOP 00??

system MySystem {
  MY_SYSTEM ,
  NOP | MY_K_DATA ,
  MY_S_DATA
}
```

The word length (i.e. the length of each schemata and the kernel) in this sample is four. Labels and the *no operation* function *NOP*, are first defined. The system declaration follows where each line successively defines the first schema, the kernel and the second schema. System definitions are given a name, here *MySystem* to be later referenced when instantiating. In this example, *MY_SYSTEM* could be a type identifier and
MY_K_DATA and MY_S_DATA would be data respectively placed in the kernel and the second schema. NOP being the nil function makes this system unable to behave as a context system. This first system would thus be stored in memory using the three words string: 0001 0001 1010.

The calculus expression corresponding to Listing 4.1 can be written:

MySystem

Adding the data information could be done by writing:

MySystem[MY_K_DATA,MY_S_DATA]

Context system definition

Listing 4.2 gives an example of a definition for a system that will behave as a context.

Listing 4.2: Definition of a context system using a word length of 4 and the first two characters of the kernel as function code

| label | MY_SYSTEM | 0001 |
| label | MY_OTHER_SYSTEM | 0010 |
| label | ANY | ???? |
| label | MY_CTX_DATA | ?10 |
| function | My_Function | 01?? |
| system | MyContext { |
| [ MY_SYSTEM , ANY , ANY ] , |
| My_Function | MY_CTX_DATA , |
| [ MY_OTHER_SYSTEM , ANY , ANY ] |
| } |

In this example, My_Function refers to the function to call when two systems interact in the current context. The use of the label ANY in the template indicates that here the value of the kernel and the right schema do not matter in the search for systems to interact with. The other use of the wildcard is shown when combining My_Function and MY_CTX_DATA using the operator ’|’.

The context system MyContext defined above would be translated as shown below:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Labels translation</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ MY_SYSTEM , ANY , ANY ] ,</td>
<td>0001 , ???? , ????</td>
<td>#qzz</td>
</tr>
<tr>
<td>My_Function</td>
<td>MY_CTX_DATA ,</td>
<td>0110</td>
</tr>
<tr>
<td>[ MY_OTHER_SYSTEM , ANY , ANY ]</td>
<td>0010 , ???? , ????</td>
<td>ahzz</td>
</tr>
</tbody>
</table>
The first schema of the context would be translated at compilation into the compressed version of 0001 ???? ?????. By using code from Table 4.1, every block of three characters would be compressed into one. Therefore 0001 ???? ???? is read 000 1?? ??? ???? and the memory value of the first schema would be #qzz, which is four characters long and thus fits the context’s schema size. Similarly, the second schema 0010 ???? ???? is read 001 0?? ??? ??? and its memory value would be ahzz.

The calculus expression describing the allowable interaction defined in Listing 4.2 is:

MySystem }- MyContext -{ MyOtherSystem

**Encoding section**

The notions of compression code, call-back functions, and word length comprise generic information needed for the compilation and execution of a program. These can be defined at the beginning of the program file in an encoding section. As shown in Listing 4.3 the user provides a compression code map file with the char_map keyword, a list of plug-ins (dynamic libraries .dll, .so, etc) containing the interaction functions to be called by the virtual machine with the func_libs keyword, sets the word length with the word_length keyword and finally sets the offset range used to encode the functions (so that the VM knows where to look in the kernel for the context function reference) with the func_offset keyword. In this example, the file sc_code.map stores the compression code, the library my_plugin.dll owns the call-back functions, the word length is four and the function code is in the two first bits of the kernel.

**Listing 4.3:** An SC program’s encoding section providing a code file, a list of plugins and defining a word length of 4 and the first two characters of the kernel as function code

```plaintext
encoding {

  // Characters code map
  char_map  "sc_code.map"

  // Functions module(s)
  func_libs { "my_plugin.dll" }

  // 4 characters long kernel and schemata
  word_length  4

  // 2 bits for functions (bit 0 to 1)
  func_offset  0:1
}
```
System and scope declarations

Once all the systems have been defined, the last aim of the SC language is to allow the declaration of system instances and their scopes (reminiscent of variable declarations and function scopes in a procedural program). Since scopes are relationships between instances, they can be handled in a program body. An example, following the previous ones, is given in Listing 4.4.

The first part of the program body declares instances, one by one (e.g. universe) or in a group with an array notation (e.g. ms[1:2] that defines the two instances ms[1] and ms[2]). (Ten instances instead of two would be defined by ms[1:10].) Note that a system definition’s name (left part of a declaration) is not a type. An instance (right part of a declaration) is always a system’s instance initialised with a system definition (triple string value) previously defined and identified by the left name (e.g. MySystem, MyContext). The definition given in Listing 4.1 sets the definition MySystem as the string 0001 0001 1010. Therefore the two instances ms[1] and ms[2] are both initialised with this very string value. Note that these system values are by definition only initial values which during computation are likely to change.

Listing 4.4: Systems instantiations and scopes setup

```plaintext
program {

  // Declarations
  Universe universe;
  MySystem ms[1:2];
  MyOtherSystem mos[1:2];
  MyContext cs[1:2];

  // Scopes
  universe { ms[1:2], mos[1:2], cs[1:2] }

}
```

Any part of a system can be altered. However, having a form of typing can be useful or simply clearer in order to identify systems, especially for writing context schemata. Some inner data can thus be used by the user as a method of typing (e.g. label MY_SYSTEM in Listing 4.1).

The second part of the program body then sets the scopes between the instances. This notion of scopes refers to embedded hierarchies. An SC program is a list of systems behaving in and belonging to systems which themselves behave in and belong to others
and so on. Since SC considers everything as a system, the program is a system, the computer running a program is a system, the user is a system, as illustrated in Figure 3.2 (page 113).

A user can interact with a program in the context of a computer. As described in the previous section, the convention chosen here is to embed the program in a single entity called universe. Note that the universe is a system. Any SC program should therefore have a universe containing everything but itself and being the only system not to be contained. This universe can be defined in any manner, but since it contains everything it cannot interact by itself with the program. Therefore there is no constraint on its definition and no need for it to act as a context. However, it is the only system a user can interact with. The universe is therefore where user’s parameters, to be changed at runtime, should be placed. It is also the place where the program can output its results.

Listing 4.4 assumes a system named Universe has been defined, although having a dedicated system definition is not mandatory. In this example the universe contains the two instances of MySystem, the two of MyOtherSystem and the two of MyContext, namely everything but itself. The calculus expression describing this organisation is:

Universe ( MySystem MySystem MyOtherSystem MyOtherSystem MyContext MyContext )

Summary

To summarise, Listing 4.5 gives the full program.

**Listing 4.5: Full sample program**

```plaintext
// Encoding section defining compression code, associated plugins, // word length and function code offset
encoding {

    // Characters code map
c char_map "sc_code.map"

    // Functions module(s)
   func_libs { "my_plugin.dll" }

    // 4 characters long kernel and schemata
word_length 4

    // 2 bits for functions (bit 0 to 1)
func_offset 0:1

}
```
// Label definitions
label ANY ?????
label MY_SYSTEM 0001
label MY_OTHER_SYSTEM 0010
label MY_K_DATA ??01
label MY_OTHER_K_DATA ??11
label MY_CTX_DATA ??10
label MY_S_DATA 1010
label MY_OTHER_S_DATA 0111

// Function label codes
function NOP 00??
function My_Function 01??

// "Universe" system definition
system Universe {
    ANY ,
    NOP ,
    ANY
}

// "MySystem" non-context system definition
system MySystem {
    MY_SYSTEM ,
    NOP | MY_K_DATA ,
    MY_S_DATA
}

// "MyOtherSystem" non-context system definition
system MyOtherSystem {
    MY_OTHER_SYSTEM ,
    NOP ,
    MY_S_DATA
}

// "MyContext" non-context system definition
system MyContext {
    [ MY_SYSTEM , ANY , ANY ] ,
    My_Function | MY_CTX_DATA ,
    [ MY_OTHER_SYSTEM , ANY , ANY ]
}
// Program section defining instances and hierarchy

program {

    // Declarations
    Universe universe;
    MySystem ms[1:2];
    MyOtherSystem mos[1:2];
    MyContext cs[1:2];

    // Scopes
    universe { ms[1:2], mos[1:2], cs[1:2] }

}

The model implemented in Listing 4.5 therefore begins with the declarations and in the initial state given below using the calculus notation. The graph notation is given in Figure 4.2.

Universe
MySystem
MySystem
MySystem
MyOtherSystem
MyOtherSystem
MyContext
MyContext
Universe ( MySystem MySystem MyOtherSystem MyOtherSystem MyContext MyContext )

Figure 4.2: Graph notation for the sample program
The interactions than can take place are:

\[
\text{MySystem } \rightarrow \text{ MyContext } \rightarrow \{ \text{ MyOtherSystem } \}
\]

The result of the interaction is not given as it depends on the function \textit{MyFunction}. If for instance the function inserts \textit{MyOtherSystem} into \textit{MySystem} the notation becomes:

\[
\text{MySystem } \rightarrow \text{ MyContext } \rightarrow \{ \text{ MyOtherSystem } \rightarrow \text{ MySystem ( MyOtherSystem ) } \}
\]

Note that the names in the program do not have any meaning as much as the computation is concerned. Indeed only the string value of a system defines it. Therefore the names used in the calculus notation do not have to correspond to the program’s names and should rather have a meaning to the user. In the above expression, the definitions names were kept.

However the systems could use more global names and then use variables to give details about their string value. In this program the four non-context systems use the aforementioned form of typing with \textit{MY_SYSTEM} and \textit{MY_OTHER_SYSTEM}. They are located in the four characters of the first schema and as such could be seen as a variable that may be modified to change the system type. Therefore these systems could be called for instance \textit{system}, use a variable and become \textit{system[MY_SYSTEM]} and \textit{system[MY_OTHER_SYSTEM]} while the contexts could be named \textit{context}. Using the calculus notation, this could be written:

\[
\text{Universe } \\
\text{system[MY_SYSTEM]} \\
\text{system[MY_SYSTEM]} \\
\text{system[MY_OTHER_SYSTEM]} \\
\text{system[MY_OTHER_SYSTEM]} \\
\text{context} \\
\text{context} \\
\text{Universe ( system[MY_SYSTEM] system[MY_SYSTEM] system[MY_OTHER_SYSTEM] system[MY_OTHER_SYSTEM] context context )}
\]

and the context interaction would be written:

\[
\text{system[MY_SYSTEM] } \rightarrow \text{ context } \rightarrow \{ \text{ system[MY_OTHER_SYSTEM] } \}
\]

Note also that the array notation \[ \] in the SC language has a different meaning than in the calculus notation. In the SC language the \[ \] brackets are used to define range of values, whether for system arrays or the function code location in the kernel memory. In the SC calculus the \[ \] brackets are used to define variables as part of a system’s definition.
SC Language Compilation

The platform includes a compiler for the SC language that turns the SC source code into a bytecode for the virtual machine. This compiler is in charge of controlling the validity of the program provided and fires potential errors to the user. The potential errors are:

- Labels, functions, systems or program section are defined before the encoding section.
  
  E.g.
  
  prog.sc : Syntax error @ line 27 : “label” is not an appropriate token or should occur after encoding section
  prog.sc : Syntax error @ line 42 : “function” is not an appropriate token or should occur after encoding section
  prog.sc : Syntax error @ line 24 : “system” is not an appropriate token or should occur after encoding section
  prog.sc : Syntax error @ line 6 : “program” is not an appropriate token or should occur after encoding section

- The encoding section is closed while not all mandatory fields were defined.
  
  E.g. prog.sc : Syntax error @ line 22 Encoding section closed without defining the obligatory fields : “char_map”, “func_libs”, “word_length”, “func_offset”

- Redefinition of encoding fields.
  
  E.g.
  
  prog.sc : Syntax error @ line 12 token “char_map” found whereas “char_map” section has already been defined
  prog.sc : Syntax error @ line 15 token “func_libs” found whereas “func_libs” section has already been defined
  prog.sc : Syntax error @ line 18 token “word_length” found whereas “word_length” section has already been defined
  prog.sc : Syntax error @ line 21 token “func_offset” found whereas “func_offset” section has already been defined

- Function labels are defined out of the function code field’s bounds.
  
  E.g.
  
  prog.sc : Syntax error @ line 42 : function code value “111” of length 3 is longer than the function code range [ 0 , 1 ]
  prog.sc : Syntax error @ line 42 : function “11” is out of the bounds of the function code range [ 0 , 1 ]
• Function labels use non-binary characters
  
  E.g. prog.sc : Syntax error @ line 42 : Function code ‘0?1’ should be a binary string \{0,1\}^* of length 3

• Unknown character in compression code.
  
  E.g. Compression error @ line 70 : No code defined to code the sequence “?p?” in the word “001??p??p”

• Compression code incompatible with the word size.
  
  E.g. Compression error @ line 70 : The code is using 14 character long sequences, any word to compress must be a multiple of 14

• A rule of the wildcard or operator is not respected.
  
  E.g. prog.sc : Syntax error @ line 57 : “000?” and “??01” both define the character at position 3

• Word size not respected.
  
  E.g.
  
  prog.sc : Syntax error @ line 70 : Declaration string “00”’s size (2) does not match word size (4)
  prog.sc : Syntax error @ line 27 : Declaration string “?”’s size (2) does not match word size (4)

• Multiple uses of a system name.
  
  E.g. prog.sc : Syntax error @ line 62 : System name “MySystem” is already used

• Unexpected tokens are found.
  
  E.g.
  
  prog.sc : Syntax error @ line 5 : “zzz” is not an appropriate token or should occur after encoding section
  prog.sc : Syntax error @ line 23 : “zzz” was found where expecting “set”, “import”, “label”, “function”, “system” or “program”
  prog.sc : Syntax error @ line 79 : “Universe” found where expecting “{”
  prog.sc : Syntax error @ line 21 : “zzz” is not an encoding feature nor a closing bracket
  prog.sc : Syntax error @ line 79 : “zzz” is not a known identifier

• Bad use of special characters.
  
  E.g.
  
  prog.sc : Syntax error @ line 70 : “,” found where expecting a word
  prog.sc : Syntax error @ line 70 : “[” found where expecting separator “,”
• Arithmetic error.
  E.g.
  prog.sc : Syntax error @ line 74 : Division by zero
  prog.sc : Syntax error @ line 74 : Syntax error in arithmetic expression

• Unknown value in an arithmetic expression (e.g. a variable not set).
  E.g. prog.sc : Syntax error @ line 85 : Incorrect or unknown value : “a”

• Bad array indices (start index > end index).
  E.g.
  prog.sc : Syntax error @ line 20 : Function offset 1 is above the end offset 0
  prog.sc : Syntax error @ line 80 : Range starting index 2 is above the last index 1

• The program section is missing when the compiler reaches the end of the program.
  E.g. Syntax error @ End of program : Missing Program Section

• Instance array notation with index out of bounds.
  E.g. prog.sc : Syntax error @ line 85 : “ms[3]” is not a system instance

• Multiple uses of a system’s instance name.
  E.g. prog.sc : Syntax error @ line 80 : Instance name “ms[1]” is already used

• Multiple scope declarations.
  E.g. prog.sc : Syntax error @ line 85 : Double scope declarations - universe already has
       “ms[1]” in its sub scope

• Multiple universes were found (i.e. systems with no super system).
  E.g. Multiple universes : “universe” and “ms[2]”

• The compiler cannot find the source file.
  E.g. Program file “prog.sc” doesn’t exist

• The compiler cannot find the compression code file.
  E.g. Cannot find or open character map “sc_code.map” in the directory “maps”

• The compiler cannot write to the target file.
  E.g. Cannot write to file “dir/prog.bsc”

• Too many errors occurred and the compilation aborts
  E.g. Too many errors, compilation aborting.
Note

A complete programming reference to the SC language with all keywords and full syntax definition is provided in Appendix B.

4.3.2 Defining Interactions

As previously explained, interaction functions implementations are provided in plugins written in C++. The plugins associated with an SC program are provided in the encoding section, as shown in Listing 4.3. The following explains how to implement an interaction function in a C++ plugin.

Listing 4.6 provides a prototype for an interaction function called here Interaction usually placed by convention in a “.h” or “.hpp” file.

Listing 4.6: Interaction function declaration

```c++
bool Interaction ( SCSystem &, const SCSystem &, SCSystem & );
```

Each interaction function has three arguments, in order, the first interacting system, the context of interaction and the second interacting system. This function has access to these three systems for reading, and to the two interacting systems for writing. The context system is here protected by the `const` keyword. Note that the write protection on the context system can be lifted if necessary by removing the `const` keyword.

An interaction function also returns a Boolean value that should indicate if a computation occurred or not. The virtual machine maintains a computation counter that can be useful for the end-user to observe the progression of a model and log data over time. If computation could not be performed or was subject to a probability and did not happen, then the function can return `false` to indicate the virtual machine that the computation counter should not be incremented. Otherwise the function should return `true`.

Listing 4.7 provides the implementation associated with the above prototype. The implementation is commonly placed in a “.cpp” file. The implementation block is where systems’ strings and scopes can be modified. All the instructions present in this block represent the interaction. An interaction can therefore be a simple scope change (e.g. insert a system into another) or a bit swap but it can also be an advanced function performing high level computation (e.g. read data input from strings and process a result).
Listing 4.7: Interaction function implementation

```cpp
// Include the header of the plugin, say here named "MyPlugin.hpp"
#include "MyPlugin.hpp"

// Defines an interaction function called here "Interaction"
bool Interaction(
    SC::SCSystem & system1,
    const SC::SCSystem & context,
    SC::SCSystem & system2
) {
    /*
    Implementation code goes here.
    Function returns true if computation occurred, false otherwise.
    */
}
```

Listing 4.8 provides an implementation example for the function MyFunction defined in Listing 4.2. This function was set as the implementation function for context system MyContext that enabled systems MySystem and MyOtherSystem to interact. The implementation given here sends MyOtherSystem into the scope of MySystem. Note that this operation does not imply removing MyOtherSystem from its current scope (which could be done using another method) as a system can have several scopes.

Listing 4.8: Example of a simple interaction function injecting a system into another

```cpp
/*
   Implementation of "My_Function" called by "MyContext"
   for the interaction of "MySystem" and "MyOtherSystem".
 */
bool My_Function(
    SC::SCSystem & MySystem,
    const SC::SCSystem & MyContext,
    SC::SCSystem & MyOtherSystem
) {
    // System "MySystem" takes system "MyOtherSystem" in its scope:
    // "MyOtherSystem" joins the scope of "MySystem", however
    // this does not affect the other scopes of "MyOtherSystem"
    MySystem.Grab(MyOtherSystem);

    // Indicates that computation occurred
    return true;
}
```
The interaction defined above can be described using the calculus notation:

\[ \text{MySystem } \rightarrow \text{MyContext } \rightarrow \{ \text{MyOtherSystem} \rightarrow \text{MySystem} (\text{MyOtherSystem}) \] 

The graphical notation for this interaction is given in Figure 4.3. The fact that \text{MySystem} grabs \text{MyOtherSystem} does not mean that \text{MyOtherSystem} loses its other scopes, but only that it is now in addition in the scope of \text{MySystem}.

![Figure 4.3: Graph notation for an interaction between MySystem and MyOtherSystem in the context of MyContext where MyOtherSystem joins the scope of MySystem](image)

Full details on how to use the library developed for this platform and its possibilities are given in Appendix B.

### 4.4 Platform Distribution

The platform distribution comes with a command line compiler, a command line virtual machine and a graphical runtime environment. The command line virtual machine only allows to run and log simulations, with no interaction with it and no visibility on the model’s behaviour. The graphical runtime environment, shown in Figure 4.4, involves a graphical user interface (GUI) that allows control for the user over the execution of a model. The user can:
• Control the simulation speed to better follow the computation happening.

• Pause the simulation.

• Go back in the past by undoing steps and then run new steps.

• Check on-line the systems value.

• Modify values of the universe definition. As explained earlier, in this work the universe is where the user can interact with a model by writing on its schemata or kernel. It is also where the program can write outputs.

• Play a recorded logged simulation, stop it and run it anew from any point in time to test alternative possibilities.

• Visualise computation on-line (see Chapter 9).

A full description of the compiler commands, the virtual machine commands and the GUI is given in Appendix A.
4.5 Summary

This chapter presented the platform that has been developed for this work. It involves programming language, compiler, virtual machine and runtime environment. It is the first high level platform implementing systemic computation, with the restrictions and benefits respectively discussed in Chapter 3 and in this chapter. Programming for this platform is performed by using the dedicated descriptive SC language for the model description, and by using C++ to implement the body of the interaction functions. The SC language compiler turns a user program into bytecode that is read by the virtual machine to execute the user model. The virtual machine can be called from the command line but a runtime environment has also been developed to allow control and analysis of the models being run. The following chapters will present the models developed on this platform for this thesis.
Chapter 5

Self-adaptation within a Genetic Algorithm

5.1 Introduction

One of the most striking features observed about living systems and their evolution is the capacity of self-adaptation. Self-adaptation is the ability of a system to adapt itself according to its environment. This means that a self-adaptive system has an ability to change in response to its environment such that the same system in a different environment would change differently. This feature of natural systems can be very useful for modern applications. Search techniques can enable self-adaptation to explore large unknown search spaces by exploiting the local cues. Strategies can change to develop alternative solutions when facing new problems. This chapter thus investigates the potential for introducing self-adaptation within SC programs.

The model chosen to investigate self-adaptation is a genetic algorithm (GA) [Holland, 1975] (see Subsection 2.4.3, page 66). A GA is suitable as it is a generic approach that aims at enabling a population that adapts itself to different problems. A second interest concerns the potential in a GA for the self-adaptation of the evolution process.

Self-adaptation within the GA model is assessed with respect to the three types of evidence presented in Chapter 3. The investigation will look at how concisely a GA model is built and based upon the exploitation of which native properties of SC. To demonstrate that the GA model is fully functional by comprising a population that adapts itself to solve large search space problems it is tested against the well-known and non-trivial travelling salesman problem. Experiments will look at the degree of self-adaptation of the population, and then of the evolution process itself.
The work presented in this chapter was published in [Le Martelot and Bentley, 2009a, Le Martelot et al., 2007a].

5.2 Genetic Algorithm Model

This section presents the genetic algorithm used to investigate self-adaptation. First a regular GA model implemented in SC is provided. Features from SC and the GA model are then exploited further to turn the GA into a self-adaptive one with minimal additional code.

5.2.1 Regular Model

In Section 3.6, the systemic analysis was introduced and illustrated using a regular GA. This chapter will use the same model, summarised in Figure 5.1.

![Figure 5.1: Structure of the genetic algorithm model here involving 1 initialiser, 1 computation space, 2 operators and 4 solutions (3 initialised and 1 non-initialised being initialised). The dashed line indicates the transformation underwent by the non-initialised solution when initialised by the initialiser.](image)

The first stage of the systemic analysis, the systems identification, defined that systems can be used to represent individual solutions and individual genetic operators. The operators act as context of interaction between the solution systems where the interaction performs selection and reproduction at the same time with the pair of interacting
Chapter 5 Self-adaptation within a Genetic Algorithm

solutions. The population of solutions is therefore a set of solution systems and their evolution is performed using simply one kind of SC interaction (one type of context system). Solution and genetic operator systems are held in a computation space system. The model initially contains non-initialised solutions that are initialised by an initialiser system and then pushed inside a computation space where they can evolve.

This part of the model follows the coarse biological abstraction of conventional GAs where solutions and genetic operators respectively model individuals (in a conventional GA genotype and phenotype are the same, hence the use of genes evaluated directly by a fitness function) and reproduction (involving sexual reproduction and mutations). Then the initialisation of solutions models the initial gene pool material of a population (i.e. before any reproduction happens). Finally the computation space system models a gene pool. Using several computation spaces would create a GA with several gene pools (similar to several GAs running in parallel). Also computation spaces could potentially share systems in order to affect each other. In this study however, only one computation space is used.

In its initial state the model can be given according to the following calculus notation:

$$\text{Universe} ( \text{Initialiser}_1 \ldots \text{Initialiser}_{n_i}, \text{Solution}_1 \ldots \text{Solution}_{n_s}, \text{Computation\_Space}_1 \ldots \text{Computation\_Space}_{n_c} )$$

$$\text{Computation\_Space} ( \text{Genetic\_Operator}_1 \ldots \text{Genetic\_Operator}_{n_g} )$$

where \( n_i, n_s, n_c \) and \( n_g \) are respectively the number of initialiser, solution, computation space and genetic operator systems.

The interaction:

$$\text{Solution}[\text{Empty} ] \rightarrow \text{Initialiser} \rightarrow \{ \text{Computation\_Space} \rightarrow \text{Computation\_Space} ( \text{Solution}[\text{Genes}] ) \}$$

transforms non-initialised solutions initially marked empty (i.e. not containing any gene yet) into initialised ones containing a potential solution (i.e. genes). It pushes them in to the computation space, progressively leading to the organisation:

$$\text{Universe} ( \text{Initialiser}_1 \ldots \text{Initialiser}_{n_i}, \text{Computation\_Space}_1 \ldots \text{Computation\_Space}_{n_c} )$$

$$\text{Computation\_Space} ( \text{Genetic\_Operator}_1 \ldots \text{Genetic\_Operator}_{n_g}, \text{Solution}_1 \ldots \text{Solution}_{n_s} )$$

As soon as at least two solutions are available in a computation space, the following interaction can take place:

$$\text{Solution}[G_A] \rightarrow \text{Genetic\_Operator} \rightarrow \{ \text{Solution}[G_B] \rightarrow \text{Solution}[G_A'] \text{ Solution}[G_B'] \}$$

It is noteworthy that in this GA evolution starts before all solutions are initialised (only two need to be initialised for it to start). New material is thus introduced constantly,
along with the evolution of previously initialised solution, until there are no more solutions to initialise, as shown in Figure 5.2.

![Figure 5.2: Progressive initialisation of a genetic algorithm until there are no more solutions to initialise. In the meanwhile evolution is ongoing from the moment two solutions are present in the computation space.](image)

Any selection and evolution method could potentially be used. In this model both are performed at the same time by the genetic operator. The fitness of the two solutions are compared and several methods can be used. The treated problem being relevant to the choice of these methods they are presented below in Section 5.3.

To make sure fitness values are computed once only for each solution, each time a new solution is created (i.e. by an initialiser or a genetic operator) its fitness is calculated and stored along with the solution in the solution system.
5.2.2 Self-adaptive Model

Among the various potential genetic operators, some might be better suited at some points in time and other better suited at other points in time. Therefore when solutions interact in the right context (i.e. with the right genetic operator) at the right times, the ability of evolution to continue making progress can be improved. In other words evolution can self-adapt.

In nature, factors affecting the progression of evolution (whether part of the evolving organisms, or of the environments of the organisms), are also subject to evolution. The contexts that affect evolutionary progress may co-evolve [Whitney and Glover, 2007], giving evolution of evolvability [Colegrave and Collins, 2008].

Genetic algorithms deal with exploration and exploitation [Holland, 1975, Mitchell, 1996]. They provide a way to search new areas of a space but can also exploit prior knowledge. Depending on the state of adaptation of a population (diverse or converging) the balance needed between exploration and exploitation required for a good adaptation may change. As the contribution of an operator to exploration and exploitation can be different from one to another, the relevance of an operator to the adaptation of the population may change in time. This cannot be set statically. To improve the efficiency of such implementation and/or to avoid a trial and error configuration phase, the operators could thus be evolved to behave according to the needs of the moment.

If a new systemic analysis is performed to identify the systems, interactions and structure, it is clear that the evolution of evolvability implies new interactions and new systems. Now the genetic operator systems must interact with each other in the context of new operator adapter systems, as shown in Figure 5.3. This enables the genetic operators to evolve in parallel to the solutions they modify.

![Figure 5.3: Scheme of a self-adaptive approach for a genetic algorithm on an SC architecture: A genetic operator adapter is added to the current interaction scheme to adapt or evolve genetic operators during computation.](image)
The way to measure the efficiency of an operator (i.e. its fitness) can vary depending on the given type of problem and will be discussed for this work further below in Section 5.3. However, whatever the measure, for a genetic operator adapter to function some measures of the operators performance need to be stored. Measures being related to genetic operators it seems reasonable to store such measures within the operators themselves.

The information could thus be stored in the genetic operator systems themselves. It could be done when a genetic operator evolves solutions, which implies writing into the context of interaction. For simplicity of implementation, this could be allowed. Note that another implementation could avoid doing this by, instead of writing in the context, writing in each modified solution a note regarding the operation undergone. A note reader system making solutions and adapters interact could then analyse these notes and update the genetic operator adapters appropriately. Allowing the genetic operator context systems to be written can therefore be seen as a simplification of such potential implementation.

5.3 Implementation of the TSP in a GA

This section presents the travelling salesman problem and then provides a functional implementation of the problem for the GA model.

5.3.1 Presentation

The travelling salesman problem (TSP) is a classic problem in the fields of computational complexity theory and evolutionary computation. The problem is, given a number of cities and the distance from one city to another, to find the shortest round-trip route that visits each city exactly once and then returns to the city of origin.

Figure 5.4 ¹ illustrates the problem with a map of Germany showing its fifteen largest cities and the shortest path passing through all of them out of 43 589 145 600 potential solutions.

The search space of the TSP problem is large and therefore presents a challenging environment to assess the self-adaptation ability of the implementation.

¹Public image from the Wikimedia project
5.3.2 Implementation

Problem Encoding

The representation of the genetic information in a TSP implementation is often chosen as a string of terminals (e.g. letters), each of them standing for a city. This is the method of representation chosen here. A chromosome (i.e. solution) is therefore a route (i.e. succession of cities).

To optimise the coding, an initial city is designated and discarded from the encoded solution. Indeed, as every city has to be visited once and as all that matters when computing the distance is the order rather than the position, fixing the first city (which therefore does not have to be coded in the chromosome) does not reduce the solution space but reduces the search space and thus the problem complexity.

There are no constraints in SC about how or where data should be stored in a system. In this problem, the TSP solution is stored in the left schema, and the distance of the route in the right schema. With the kernel using the $NOP$ function, these systems will not act as context for interacting systems matching the schemata. (Note that this would be
possible should a different function be defined.) This data organisation is summarised in Figure 5.5. Note that the data, whether the list of letters (i.e. cities) or fitness values are encoded in binary in the machine code.

<table>
<thead>
<tr>
<th>Schema 1 (left)</th>
<th>Kernel</th>
<th>Schema 2 (right)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution (e.g. AECDB)</td>
<td>NOP</td>
<td>Solution’s fitness value</td>
</tr>
</tbody>
</table>

**Figure 5.5:** Data organisation within a solution system for a GA solving the TSP

### Genetic Operators

In this implementation, genetic operations are performed by the interaction function of the genetic operator systems. Here, several types of genetic operators using different approaches for selection, reproduction and mutation are used.

Selection is performed using a tournament selection size 2. The fitness of both solutions is compared and the winner can be selected according to two alternatives:

- **Strictly elitist (or keep best KB):** the best solution is always kept. The fitness value used to evaluate solutions is their path length. The best solution is therefore the one that describes the route having the shortest length.

- **Fitness proportional (FP):** the better a solution, the more likely it is to be kept (roulette-wheel selection). The fitness used to evaluate a solution \( s_1 \) competing with a solution \( s_2 \) is:

\[
\text{fitness}(s_1) = \frac{\text{path}_\text{length}(s_2)}{\text{path}_\text{length}(s_1) + \text{path}_\text{length}(s_2)}
\]

Solution \( s_2 \) has the fitness:

\[
\text{fitness}(s_2) = 1 - \text{fitness}(s_1)
\]

A random number within the interval \([0,1]\) is then drawn. If it is lower than or equal to \( \text{fitness}(s_1) \) then \( s_1 \) is kept, otherwise \( s_2 \) is kept.

It is noticeable that such configuration has many similarities with a steady-state GA [Whitney and Kauth, 1988]. However, the selection here chooses two competing solutions (systems) at random rather than from a fitness sorted set. (It would be possible to sort and order solutions using sort systems and linking systems to produce an ordered chain of solutions, but this was deemed unnecessary complexity here.)
As a result of selection, the selected parent is kept unchanged and reproduction replaces the other solution with a new solution, created either by crossover using the two interacting solutions as parents or by duplication and mutation of the selected solution. This interaction is described by the following calculus notation:

\[
\text{Solution}[G_A] \xrightarrow{\text{Genetic Operator}} -\{ \text{Solution}[G_B] \} \quad \rightarrow \quad \text{Solution}[G_A] \text{ Solution}[G_C] | \text{ Solution}[G_B] \text{ Solution}[G_C]
\]

In the TSP, chromosomes are ordered lists of unique genes (cities). Therefore a swap of genes could create duplicates while removing other genes and thus create invalid solutions. Several crossover methods have been investigated for this problem and in this work two different crossover methods were chosen:

- **Partially mapped crossover (PMX)** [Goldberg and Lingle, 1985]: A chunk (or several) from a first parent chromosome is copied as it is to a new chromosome, the rest is filled in order from the other parent swapping the genes already present in the copied chunk section with the corresponding genes in the chunk section from the other parent (mapping).

  E.g. Crossing ABCDEFGH and HCDBEGAF, taking the first chromosome as first parent and cutting between genes 3 and 6 would give HACDEFGA:

<table>
<thead>
<tr>
<th>Parents</th>
<th>Mid-step &amp; Mapping</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEFGH</td>
<td>HCCDEFAF</td>
<td>HBCDEFGA</td>
</tr>
<tr>
<td>HCDBEGAF</td>
<td>C → D → B, F → G</td>
<td></td>
</tr>
</tbody>
</table>

- **Ordered crossover (OX)** [Davis, 1985]: A chunk (or several) from a first parent chromosome is copied as it is to a new chromosome, the rest is filled in order from the other parent but jumping the genes already present.

  E.g. Crossing ABCDEFGH and HCDBEGAF, taking the first chromosome as first parent and cutting between genes 3 and 6 would give HBCDEFGA:

<table>
<thead>
<tr>
<th>Parents</th>
<th>Mid-step</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEFGH</td>
<td><strong>CDEF</strong></td>
<td>HBCDEFGA</td>
</tr>
<tr>
<td>HCDBEGAF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These crossover operators were chosen because they are commonly known and usually provide good results. They both guarantee to create only valid chromosomes and to keep the order in the exchanged chunks of genes.

When duplication and mutation is used to generate the child, several mutations can be considered. However due to the nature of the chromosomes, any form of mutation (e.g. randomly changing a gene) is not possible. The uniqueness of each gene needs to be asserted. Three alternative methods of mutation were chosen:


- **Swap**: chooses two successive genes and swaps them.
  
  E.g. Mutating ABCDEFGH at gene 3 gives ABDCEFGH.

- **Move**: chooses a gene, moves it to another location, and brings the gene from this other location to the location of the first chosen gene.
  
  E.g. Mutating ABCDEFGH at genes 3 and 7 gives ABGDEFCH.

- **Reverse**: chooses a route portion and reverses it.
  
  E.g. Mutating ABCDEFGH between genes 3 and 7 gives ABGFEDCH.

In this implementation an operator for each combination of selection and evolution method (i.e. KB or FP using PMX, OX, Swap, Move, Reverse) is defined. Therefore, the context function of each operator defines one of the ten possible combinations:

\[
\begin{array}{ccc}
\text{PMX} & \text{PMX} \\
\text{OX} & \text{OX} \\
\text{KB} \times \text{Swap} & \text{FP} \times \text{Swap} \\
\text{Move} & \text{Move} \\
\text{Reverse} & \text{Reverse}
\end{array}
\]

The genetic operator context is therefore a major interaction in the GA implementation. The pseudo-code of its interaction function is given in Algorithm 5.1.

**Algorithm 5.1** Pseudo-code of a genetic operator’s function within a GA for the TSP

Read the fitness of each of the two solutions from the systems memory

if selection method is Keep Best then
  Select the solution with worse fitness to be replaced
else
  Select the solution to be replaced using a roulette-wheel
end if

if operator is a crossover then
  Compute a new solution as a recombination of the two solutions
else
  Compute a new solution copying the best fitness solution and performing a mutation
end if

Replace solution selected for replacement
Compute the fitness of the newly created solution and write it in its system

**Population Size Estimate**

The choice of the number of solutions, or population size, has been addressed in [Alander, 1992], and later specifically for the TSP using a GA in [Julstrom, 1996]. In the latter, for a problem with \(n\) cities and exactly one optimal tour (more optimal tours make the
problem easier), the aim is to find the minimum population size $s$ which has at least a probability $p$ that any population made of $s$ randomly selected tours includes all the edges of the optimal tour. $s$ is defined by Equation 5.1.

\[ s \geq \frac{\ln(1 - \sqrt[n]{p})}{\ln\left(\frac{n-2}{n-1}\right)} \] (5.1)

The population size depends on the problem size. It has also been shown in [Julkstrom, 1996] that the population size in a GA for TSP is not critical, especially if the algorithm includes mutation, and observed that it is not clear what the best population size should be.

In this work the population size estimate from [Julkstrom, 1996] and given in Equation 5.1 is used, taking $p = 0.99$.

**Problem Size**

To assess the TSP implementation a well known city map of 48 cities (gr48) from the TSPLIB\(^2\) was chosen. The best solution for this map is empirically known and has a length of 5046. Knowing the best solution is useful for it helps to gauge the quality of the solutions the GA implementation finds.

The population size to use was then found using Equation 5.1 and setting $n = 48$ and $p = 0.99$:

\[ s \geq \frac{\ln(1 - \sqrt[48]{0.99})}{\ln\left(\frac{46}{47}\right)} = 194.8128 \]

therefore

\[ s = 195 \]

A population size of 195 solutions was hence used.

**5.4 Experiments and Analysis**

**5.4.1 Experiments**

To demonstrate the self-adaptation ability of SC with the GA implementation, four series of experiments were conducted. Experiments test the various selection methods, genetic operators and the genetic operator adapter to provide the best self-adaptation ability first at the population and then at the evolution level. All setups involve:

\(^{2}\text{http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/}\)


- 1 universe (like in all SC programs),
- 1 computation space,
- 1 initialiser,
- 195 solutions,
- \( p \) genetic operators: this will vary upon the experiment.
- \( a \) genetic operator adapters: only in the last experiment.

Crossover operations are performed using two points. Mutations are applied one at a time (i.e. a mutation operator can only apply one mutation per interaction). 8 bits are used for each gene. There are thus \( 2^8 = 256 \) possible genes. In the given problem there are 48 cities and the initial city is not encoded. The problem therefore needs only 47 different genes and thus 8 bits per gene is sufficient. A solution is \( 47 \times 8 = 376 \) bits long. Solutions are stored on the first schema and the chosen word length is 2048, which is therefore large enough. (Such configuration is appropriate for dealing with a problem up to 257 cities: 256 cities to encode, requiring 8 bits per gene and \( 256 \times 8 = 2048 \) bits per solution.)

**Experiment 1: Selection Methods Comparison**

For adaptation to take place in a population, the selection process needs to select on average the fittest individuals to enable reproduction to exploit them mainly. The aim of the first experiment is to assess relative adaptation performances of the two selection methods in this implementation.

Three SC programs were created, one using only keep best (KB) selection, one using fitness proportional (FP) selection, and one using both. For each selection method, one operator of each type was used. The experiment was repeated 10 times, with consistent results. Representative results of one run from this experiment are shown in Figure 5.6.

As can be seen in Figure 5.6, while the use of KB enables solutions to converge in a normal manner towards better solutions, the FP method seems to actively prevent evolution, with evolution halting early on. When both operators are used, performance is slightly improved compared to FP, but again evolution halts early.

This result is not surprising if considering the mechanisms behind the selection methods. Systemic computation by default causes solutions in the same scope (here computation space) to interact randomly in the context of a genetic operator system. Although only two individuals compete at a time, the overall effect on the population is similar to fitness ranked proportional selection. While the KB operator acknowledges this, the FP
selection contests it partially. When selecting with FP, if out of the two solutions one is much better than the other, the FP selection will tend to act as a KB selection. If the two solutions are close, the FP selection will be fairly close to random, and randomness is already present when picking the two interacting solutions. It thus appears that the FP operator can only slow down the convergence process without bringing much in return. For this reason, only the KB method is used in further experiments.

**Experiment 2: Genetic Operators Comparison**

In addition to selection, gene diversity depends on the method of reproduction. To have a good population adaptation, diversity is important. Indeed, if operators for example tend to keep and duplicate the best solutions, the population will quickly tend to contain only copies of these best solutions and may converge prematurely. The adaptation may then be far from optimal.

The aim of the second experiment is thus to evaluate and compare the efficiency of the operators. Using KB selection, six versions of the program were created, each using one of the five operators, and one using all five together (one copy each). Each was executed 10 times with consistent results. Figure 5.7 provides a plot of one run showing representative results.

Examining both speed of optimisation, and the ability of the population to keep improving without premature convergence, it can be observed in Figure 5.7 that all operators
Figure 5.7: Comparative test of the operators all together and individually together perform better in time than any used individually. When used alone, both crossover operators seem to lead to premature convergence. Figure 5.8 shows the effectiveness of OX and PMX over time, by plotting Equation 5.2 over one run involving OX alone and another run involving PMX alone.

\[
effectiveness = \frac{\text{best parent's fitness}}{\text{child's fitness}} \tag{5.2}
\]

Both settle at 1 indicating children with identical fitness to parents. However even when reaching solutions difficult to improve, the children would then often be less good than their parents and the ratio would be different than 1. Having a constant ratio set to 1 with diversity in the genes is therefore not probable and suggests that premature convergence of the population to a non-optimal solution is occurring. In the absence of mutations, crossover progressively selects and copies good portions of genes to the detriment of diversity. As a consequence, crossover should not be used alone.

When analysing results from Figure 5.7, only the reverse mutation seems to enable rapid convergence and continuous improvement thereafter. However, when all operators are available together, the population evolves more effectively.

Indeed, when the contribution of each operator is analysed in more detail by assessing how frequently each operator produced a fitter solution over time, it becomes apparent that all enable convergence to solutions at different rates, and that those rates all vary
Surprisingly maybe the swap operator is providing on average more fitter solutions than any other operator while Figure 5.7 showed it is the slowest operator with respect to convergence. Also, the reverse mutation operator, even though providing a faster convergence than other operators does not provide more fitter solutions on average than crossover operators or the swap mutation. The move mutation hardly produces fitter solutions. The two crossover operators tend to provide a bit more fitter solutions than worse solutions over time.

Swap being a very local operator, it is not exploring the search space and is just exploiting the current solutions. It is thus unlikely to converge quickly but it is able to thoroughly explore and improve the current solution. The two crossovers are rather exploring operators. The reverse mutation seems to be in between. Even though working locally, it can when repeated quickly create sequences very different from the original ones, sending groups of genes away from their initial location. As it is a mutation, it does not have the drawback of the lack of diversity observed with crossovers. This can explain why it performs much better in time than crossovers even though when competing together its fitness is lower.
Experiment 3: Genetic Operators Combination

In order to improve the speed of adaptation of the population, knowing the most efficient operators individually is a first step but knowing efficient combinations can provide better results. Observations from experiment 2 can provide insights regarding what operators can work well together. The swap operator can only be useful but cannot be enough. Crossovers can only work with a mutation. PMX will be selected over OX as it showed better performance. Reverse mutation is the best for convergence just on its own but it nevertheless needs an additional contribution to get better than all operators together. The move mutation does not provide anything really interesting in this configuration.

Another experiment involving the PMX crossover, the reverse and the swap mutations combined in various manners was performed. Each configuration was run 5 times with consistent results. Performance of the various combinations over a representative run are given in Figure 5.10.

Results show that the reverse operator is needed in the computation since the PMX-Swap configuration, like the Swap alone, progresses very slowly. PMX-Reverse seems is a bit slower than Reverse-Swap and the addition of PMX to the latter does not improve the results. Therefore, a simple and efficient configuration can thus just use reverse and swap mutations together, without any crossover.
Experiment 4: Self-adaptation of Evolution

As suggested by Figure 5.9 different combinations of operators would be more appropriate at different times during evolution and could potentially improve the speed of adaptation of the population even more. To improve the adaptability of a population, the evolution process is made adaptive as well.

Equation 5.2 showed how the fitness of individual operators could be assessed by comparing the relative fitness of parent and child solutions produced by an operator. This information is calculated when a genetic operator processes a pair of solutions. This measure will be used as operator fitness for the genetic operator adapter and stored within it when evolving solutions.

The self-adaptation of evolution feature is implemented using a genetic operator adapter which can adapt, therefore modify, the operators’ type over time depending on their performance. When two operators interact in the context of this adapter system, a roulette-wheel algorithm is used. A fitter operator has a stronger probability of replacing a less fit one, thus making the number of fitter operators more numerous in the population.

The average fitness of a genetic operator is measured within a window of \( w \) last operations. This size grows in time following the distribution:

\[
w(n) = a \cdot (1 - e^{-k \cdot n})
\]
where \( n \) is the total number of computations performed and \( a \) and \( k \) are constants respectively set to 1000 and 0.0001. This distribution is illustrated in Figure 5.11.

Lack of diversity was penalised by giving 0.9 instead of 1 when an operator generates an identical copy of the chosen solution.

The GA model was run 5 times using 3 instances of each of the 5 operators, using KB selection, and 1 operator adapter. Figure 5.12 shows a representative run, comparing
the adaptive algorithm with 4 other combinations of operators. The results show that the adaptive method consistently outperforms the other approaches. Analysis indicates that the relative number of operators changes during the course of evolution.

5.4.2 Analysis

Experiments showed for the travelling salesman problem that populations adapt faster under an elitist selection (KB) than under a fitness proportional selection (FP) and favour large mutations (Reverse) over local mutations (Swap, Move) and crossovers (PMX, OX).

Enabling the self-adaptation of evolution then allowed all operators to potentially be used together based on their contribution to self-adaptation within the population. Experiments showed that self-adaptation at a meta level (i.e. evolvability) can improve self-adaptation at the lower level (i.e. population).

Section 3.8 (page 129) presented the three types of evidence that would be collected in this thesis and the contribution of this model to each of these evidence is now discussed.

Improved Naturalness

Natural computation is built upon stochastic, distributed and local computation (see Subsection 2.2.5, page 48). Biological evolution defines a process gradually emerging from the adaptation of individuals [Darwin, 1859]. Yet, most of the traditional GA implementations involve a global algorithm that sorts the population, selects the fittest individuals and creates the next generation out of them (see Algorithm 2.1, page 68).

This global approach is not natively suggested by the SC rules which emphasise on local interactions between entities of a model. (Here breeding individuals reproduce in a context of survival of the fittest where offsprings result at a lower level from chromosome crossover and mutation.) The GA following these rules very closely is performing evolution in a stochastic and distributed way through local interactions only (i.e. no global sorting). This approach induces a closer resemblance to biological evolution. Experiments 1 to 3 showed that such design is valid and can solve complex tasks like the travelling salesman problem.

Furthermore, natural biology is able to evolve evolvability by modifying the effect of crossover [Miick et al., 1997] and mutation [Rodríguez-Trelles et al., 2000]. The likelihood of a gene mutation or the way a crossover is performed can thus be affected by its context, whether biological or external.
While the notion of genetic operator is inherited from the GA literature, it can be seen as the context encompassing all those factors that can affect the reproduction, including crossover and mutation. Modifying those contexts affects the lower-level interactions taking place. This has been modelled by adding a genetic operator adapter comparing two by two genetic operators. The evolution of operator contexts is however driven by their ability to evolve solutions. The naturalness of the model is therefore limited by the GA concept but the model remains relevant from a biological perspective. Experiment 4 showed that such design is valid and can solve the task of enhancing the adaptation speed of a population of solutions to the travelling salesman problem.

Conciseness, Compactness and Readability

The SC genetic algorithm model is made of only 4 dedicated systems in total (the universe being present in any SC program), including 2 kinds of context system, while encompassing the three concepts of solution initialisation, addition to a gene pool and evolution. This compact model is due to the exploitation of the natural rules of SC and their blending into the core of the algorithm:

- Solutions and genetic operators are just systems.
- The stochastic and distributed computation is inherent to SC.
- Evolution is simply performed by one form of interaction.
- Evolution self-adaptation can be performed by adding one single context system type.

Therefore much of the material for building a genetic algorithm (stochastic, distributed and local computation) is natively provided by SC, as opposed to conventional approaches (natively deterministic and centralised, see Subsection 2.2.3, page 41), thus resulting in a simpler and more concise implementation.

Also, the specific rules of SC imposing every entity as a system and using clearly defined contexts for their interactions along with the conciseness of the model makes the model more readable than conventional implementation (e.g. no explicit loop, no solution picking process).

Properties Exploitation Ability

The GA model, by exploiting stochastic, distributed and local knowledge properties provided the behavioural property of self-adaptation at the population level, as illustrated in Figure 5.13.
Figure 5.13: Properties exploitation diagram for a genetic algorithm implemented in SC. The combination of distributed and stochastic computation with local interactions leads in the implementation of a GA to a self-adaptive process. Solution systems host potential solutions to a problem while genetic operator systems host the selection and reproduction functions. Adding genetic operator adapters to the self-adaptive process enables a meta level of self-adaptation.
A traditional implementation (e.g. C, Java) needs to simulate the stochastic selection of solutions. This simulation is commonly part of a global sorting and selection process which rules out the fully distributed and local computation property (see Algorithm 2.1, page 68).

Evolution on the contrary is a distributed process resulting from local interactions. In SC each system is an agent able to randomly interact with one other agent at a time only. The SC implementation therefore natively provides stochastic interactions and imposes distributed systems with local interactions only. (Global computation can be simulated, but out of systems performing local interaction.) These native properties can therefore be used as building blocks to obtain self-adaptation out of solution and genetic operator systems undergoing SC computing.

Then, by reusing the same native computational properties and the population’s self-adaptation property, a higher (meta) level of self-adaptation could be introduced to enable a self-adaptive evolutionary process. In Figure 5.13 this is illustrated by adding the genetic operator adapter to the self-adaptive process, thus resulting in self-adaptive and meta self-adaptive processes.

5.5 Summary

This chapter presented an investigation of self-adaptation using a genetic algorithm implementation on the SC platform. The implementation exploited the rules of SC to create a concise GA. This GA is based on natural rules only with solutions evolved one at a time thus without any concept of all new generation, and using a one to one comparison thus ruling out any global sorting. The self-adaptation ability of the GA population was tested and experimentally demonstrated against the non-trivial travelling salesman problem.

In order to improve its self-adaptability the GA was then turned into a meta self-adaptive one with minimal additional code, by exploiting again the native rules of SC and building a meta-GA within the GA. The GA could then evolve its evolvability.

This work also illustrated the natural and concise modelling SC offers. Flexibility of model evolution was demonstrated showing how a program can be improved through exploiting its properties and SC’s further by topping it up with additional systems rather than by redesigning it.

The next chapter exploits further the GA model and explores the property of fault-tolerance.
Chapter 6

Fault-tolerance within a Genetic Algorithm: from Crash-proof Computing to Self-repair

6.1 Introduction

Reliability in computer or engineering systems is undoubtedly a key requirement in the development process. Safety within critical control systems, and reliable data transfers, require tolerance to unexpected and unwanted phenomena. With the increasing performance, potential and complexity in machines and software, it has however become increasingly difficult to ensure reliability in systems. Software regularly crashes, top of the line robots break down on the wrong kind of ground, power distribution networks fail under unforeseen circumstances [Bentley, 2005a]. Yet, there are many approaches to limit potential failures.

In nature, old and potentially damaged cells are constantly being replaced by new ones, DNA is able to repair and replicate with error control [Darnell et al., 1990]. The lifespan of cells is shorter than the life of an organism, so fault-tolerance and self-maintenance are essential for the survival of the organism. The failure of some components does not destroy the overall organism; cell death is an important part of staying alive.

To ensure durability, fault-tolerance must therefore be as mandatory in a system as its ability to solve a given problem. The latter could actually hardly be trusted or even possible without the former. Conventional computers are examples of non fault-tolerant systems where the smallest error in code, corruption in memory, or interference with electronics can cause terminal failures [Bentley, 2005b]. For best reliability, computational
systems could mirror biological systems. Biology has often been a successful inspiration in computation (e.g. artificial neural networks, genetic algorithms, ant colony optimisation) although conventional computation differs widely from natural computation.

In this respect this chapter addresses the property of fault-tolerance. Fault-tolerance can be achieved with various levels of tolerance to faults and involving various features, from crash-proof computing to the self-repair ability. This work investigates all these aspects.

As was discussed in Section 3.5 (page 122), SC programming involves a list of agents (the systems) defined and declared in an initial state which are then executed by letting them behave indefinitely and stochastically. The outcome of an SC program is therefore the result of many interactions leading to an emergent behaviour rather than a deterministic predefined one.

Programming with SC has various benefits regarding fault-tolerance:

- A program relies on many independent systems and the failure of one of them cannot destroy the whole program. Like cells in biology, one system collapsing or making mistakes can be compensated by other systems working correctly. The behaviour of a program is therefore not necessarily compromised by the failure of some software components.

- The continuous and parallel computation of SC means that a failed interaction does not prevent any further interactions from happening. The behaviour of a program is therefore not necessarily compromised by a non-expected situation within the software.

- SC does not permit memory corruption, and even if individual systems contained fatal errors (e.g. divide by zero) the whole program would not halt. Every SC program is already in an infinite, never-ending loop so it cannot crash in the conventional sense, just as natural systems that do not have such concept.

- Having multiple instances of similar systems not only provides redundancy, it also makes it easy to introduce a self-maintenance process which allows similar systems to fix each other, including the self-maintenance systems themselves.

These ideas are illustrated with another implementation of a genetic algorithm on the SC platform (see Chapter 5). While any program could be used for the demonstration of fault-tolerance and self-repair, a GA was chosen as its natural parallelism simplifies the implementation in SC. The investigation will assess how naturally SC demonstrates crash-proof computing and fault-tolerance and by which properties exploitation. The
behavioural property of self-maintenance is then considered in terms of how naturally and concisely it can be obtained, and again based on what properties. Experiments will assess the achievement level of each property.

The work presented in this chapter was published in [Le Martelot and Bentley, 2009a, Le Martelot et al., 2008a].

6.2 Testing Model

6.2.1 Genetic Algorithm Model

In Chapter 3 and Chapter 5 a genetic algorithm (GA) was presented and used. This chapter will use a similar model.

In this investigation the GA program is assessed for fault-tolerance. To challenge the model as much as possible it will be fully autonomous from the initialisation of solutions to the delivery of the best found solutions. The best found solutions will be sent to the universe which will keep note of them in its data. The user can then read from there the state of progression of the GA. (In other models this delivery could be omitted by outputting the found solutions to a terminal for instance when operators evolved solutions.)

To transfer data from the program to the user using this method another system not present in the other GA models and able to provide a relevant output is needed. It can be done by introducing a solution transfer system comparing the solution within a solution system with the solution currently available for the user from the universe’s data. If the solution system contains a fitter solution then this solution replaces the one currently being held. This interaction is described below:

\[
\text{Universe[known]} \rightarrow \text{Solution Transfer} \rightarrow \text{Solution[new]} \rightarrow \text{Universe[new] Solution[new]} \mid \text{Universe[known] Solution[new]}
\]

To guarantee that the universe and solutions can interact within a solution transfer context, the solution systems as well as the solution transfer systems are located within the universe. Therefore, when solutions are pushed in a computation space by the initialisers, they also remain within the universe (systems can have several scopes). This is another difference with the GA models previously presented in this thesis.

The GA will therefore go from the initial organisation:

\[
\text{Universe( Initialiser}_1 \ldots \text{Initialiser}_{n_i}\text{ Solution Transfer}_1 \ldots \text{Solution Transfer}_{n_t}\text{ Solution}_1 \ldots \text{Solution}_{n_s}\text{ Computation Space}_1 \ldots \text{Computation Space}_{n_c})
\]
Computation_Space ( Genetic_Operator_1 ... Genetic_Operator_n_g )

towards the organisation:

Universe( Initialiser_1 ... Initialiser_n_i Solution_Transfer_1 ... Solution_Transfer_n_t Solution_1 ... Solution_n_s Computation_Space_1 ... Computation_Space_n_c )

Computation_Space ( Genetic_Operator_1 ... Genetic_Operator_n_g Solution_1 ... Solution_n_s )

where n_i, n_t, n_s, n_c and n_g are respectively the number of initialiser, solution transfer, solution, computation space and genetic operator systems.

Figure 6.1 illustrates the global organisation of the GA, using only few systems for readability.

**Figure 6.1:** Structure of the genetic algorithm model here involving 1 initialiser, 1 computation space, 1 solution transfer, 2 operators, 4 solutions (3 initialised and 1 non-initialised).

Finally, the GA needs an aim. In order to observe its progression in a visually convenient way, here the given objective is simply to evolve a string of bits that matches a target pattern: a bits string of 256 1s. Note that any other fitness function could have been used instead.

Because the problem to solve and its encoding are different than in Chapter 5, the selection and evolution methods performed by genetic operators are also different from
the previous model. There is only one kind of genetic operator. It selects the less fit solution out of the two and performs one of these two operations:

- Replace it with an offspring of the two solutions created by a two-points crossover.
- Apply it a mutation rate of 0.01 bitwise probability.

6.2.2 Simulating faults

The first aim of the study is to assess the fault-tolerant behaviour of SC programs. To achieve this, faults first have to be modelled. Hardware or software faults can be simulated by randomly altering a memory state (replacing its value with any possible value) with a given rate. By this, unpredictable random mistakes that can occur anywhere at any time in the program are provided.

On the platform used in this work, systems are strings of characters (characters are '0', '1', '?' and compression code values), with each character encoded in binary using one byte (8 bits). Therefore a memory fault can alter any byte value leading to any possible value between 0 and 255. Considering a 10 characters string with the value 000111??11, after a fault occurring on the second bit the value could be for instance 0#0111??11. If this character is part of some numeric data, the data is corrupted. If this character is used to identify the system for interaction, it can no longer be identified properly. Considering now a context system schema, a corrupted character makes the context unable to identify properly an eligible system. On the kernel of a context, damaging the function code would prevent the correct interaction function from being called. Note however that in this implementation the interaction functions code (located in the plugins) is not subject to corruption as only the SC code is corrupted.

Faults should be modelled so that their systemic existence (i.e. the fact that the systems involved in their modelling exist) does not disrupt the inner organisation of the program. In other words, if the fault simulation systems are introduced in the program with a null fault probability, the program should behave perfectly normally as if the fault systems were absent.

A fault can be due to an unexpected phenomenon that interacts with a component. Whether this phenomenon is an increase of temperature leading to a hardware failure or a programming mistake in memory addressing, the result is the alteration of the memory state in the context of the laws of physics that made this physical change possible. Any program system is therefore susceptible to faults, whether software or hardware initiated.
In SC modelling, the above can be achieved by putting any program system (i.e. system part of the initial program) within a *phenomenon system* also containing the *laws of physics*, as shown in Figure 6.2. A fault is indeed due to a phenomenon that is specific to each fault and each fault occurs within the scope of such phenomenon. Then, whatever happens is undergoing the laws of physics of our universe. The unexpected phenomenon therefore interacts with a program system within the context of the laws of physics (the same laws of physics system is within the scope of all phenomena). Note that the laws of physics being represented by one unique system, only one system can be corrupted per cycle.

![Figure 6.2: Interaction between an external unexpected phenomenon and a program system in the context of the laws of physics and within the scope of the phenomenon (i.e. the system is encompassed in the field of interaction of the phenomenon).](image)

In the case of the GA program, a program system can therefore be a computation space, a solution, an operator, a solution transfer or an initialiser.

The user provides parameters to the universe and reads what the program returns from it. Also, the phenomena and the laws of physics are not part of the *tested program*. Therefore the universe, the laws of physics or the phenomena are not considered as fallible components.

### 6.2.3 Implementing Self-maintenance

As previously discussed, biology partly relies on redundancy to provide fault-tolerance and self-repair. While redundancy offers the potential for fault-tolerance, a program being regularly damaged, even if fault-tolerant, would stop working properly at some point unless repair is provided to make it recover. To delay or potentially avoid such lethal degradation, a program could therefore be repaired. An elegant way to have a program repaired would be an *on-line* self-maintenance of the program. The program would repair itself. No external intervention would then be required as the program would show a homoeostatic behaviour.
System definitions in the program can be instantiated several times. Therefore, interacting systems instances could try to fix each other in a self-maintenance context, as shown in Figure 6.3.

![Figure 6.3: Two program systems interacting within a context of self-maintenance](image)

Indeed, if the two systems are similar on their healthy parts, then they can replace the damaged parts of each by the ones of the other if these are healthy. The self-repair ability of the program then arises from its conception in independent and multiple times instantiated systems. For this reason, the self-maintenance context systems should also be instantiated several times (as they can be damaged as well). The more redundant the information (the more duplicated systems) the more likely systems are to be able to fix each other and the more likely the function they play in the program is reliable. The addition of the self-repair ability also illustrates again the flexibility within SC models where new features can be provided by injecting new systems into an existing model.

### 6.3 Experiments and Analysis

#### 6.3.1 Experiments

Previous work [Bentley, 2005b] showed that programs evolved using fractal gene regulatory networks cope better with code damage than human-designed or genetic programming generated programs. The following experiments focus on how human-designed programs for SC can natively cope with code damage.

In the following, `nb_iterations`, `nb_systems`, `nb_contexts` and `nb_simple` respectively refer to the number of iterations, systems, context systems and non-context systems. In all experiments, the solutions are 256 bits long. The best solution’s fitness is thus 256. To fit solutions within the systems a word length of 256 characters was chosen. The total length of a system is therefore:

\[
l_s = 256 \times 3 = 768
\]
All runs for a particular configuration are repeated 10 times, and the presented results are averaged over the 10 runs.

When setting up an experiment, to appreciate the damage that will be inflicted to programs, an estimation of the quantity of errors randomly introduced in the programs can be produced. Given a character-wise fault probability \( p_c \), the following values can be calculated:

- \( p_s = 1 - (1 - p_c)^{ls} \): system corruption probability
- \( q = p_c \cdot ls \cdot \frac{nb_{iterations}}{nb_{contexts}} \): estimated quantity of corrupted characters over an execution

The ratio \( \frac{nb_{iterations}}{nb_{contexts}} \) represents the number of times each fallible context system can attempt an interaction. Indeed this ratio gives the number of cycles (see Section 4.2) the SC virtual machine would have executed and each context can attempt one interaction per cycle. The number of iterations a program needs to finish depends on its configuration. Therefore for each experiment an average value that experiments showed to be required for the program to finish is provided. This value can then be used to estimate the damage the same program will undergo when introducing faults. (Note that the estimation of the number of cycles requires, in order to be correct, the number of context systems to remain constant during execution. In this program it does not change. It could only potentially change when introducing faults and in the event when successive introduced faults turn a context system’s function into NOP. However the measure is performed on a non-corrupted program.)

Also, damages made to context systems have a stronger impact although as likely to happen as for any other system. Indeed other systems can be scopes and hold no or little data; or data systems usually using less crucial characters than the context systems. It is therefore useful to estimate the quantities \( q_c \) and \( q_s \) of damage that will be respectively made to fallible context and simple (non-context) systems:

- \( q_c = q \cdot \frac{nb_{contexts}}{nb_{systems}} \): Number of context system characters corrupted in one run
- \( q_s = q \cdot \frac{nb_{simple}}{nb_{systems}} \): Number of simple (non-context) system characters corrupted in one run

The first experiment assesses the behaviour of a naive GA program instantiation when facing faults. The second experiment then investigates the impact of systems redundancy in the GA instantiation. In the third experiment, the faults level is raised to inflict severe damage and challenge the limits of systems redundancy. Finally the last experiment assesses the self-repair ability of the program when facing such severe damage.
In all experiments, the curves show the average progression of the algorithm over 10 runs. Consistency in the results was observed, hence making the amount of runs sufficient and their average representative of the algorithm’s behaviour.

**Experiment 1: Without System Duplication**

Program setup with a minimalist configuration (contained within a universe system):

- 1 initialiser,
- 1 computation space,
- 1 solution transfer,
- 25 solutions,
- 1 crossover operator,
- 1 mutation operator.

In this configuration: $nb_{contexts} = 4$ and $nb_{systems} = 30$.

10 runs of the program were performed with no fault and 10 runs were performed with faults injected with character-wise fault probability $p_c = 0.0001$ giving the system corruption probability $p_s = 0.0739$.

Experimental measures give $\frac{nb_{iterations}}{nb_{contexts}} \approx 3700$, which leads to the quantities:

- $q(p_c = 0.0001) = 284.16$
- $q_c = q \cdot \frac{4}{30} \approx 38$
- $q_s = q \cdot \frac{26}{30} \approx 246$

The estimation of damage is of about 284 characters damaged during the program execution, distributed as 38 characters (13.4%) of context systems and 246 characters (86.6%) of non context systems.

Figure 6.4 shows the program progression with and without faults.

This first experiment shows that the GA stops evolving at a very early stage when its program is corrupted by faults. However, it is noteworthy that when injecting faults, the program does not crash; it merely stops evolving properly. The systemic computer is crash-proof as systems deterioration can only stop or corrupt individual interactions, but
as the whole program consists of systems interacting in parallel, the other uncorrupted individual interactions will continue as normal.

**Experiment 2: With System Duplication**

The previous program was performing correctly for a very short time due to the single instantiation of all the systems (except the solution systems since a GA by definition uses a population of solutions).

The second experiment thus uses duplicated systems:

- 5 initialisers,
- 3 computation spaces,
- 10 solution transfers,
- 25 solutions,
- 10 crossover operators,
- 10 mutation operators.

In this configuration: $nb_{contexts} = 35$ and $nb_{systems} = 63$. 
Like in experiment 1, 10 runs of the program were performed with no fault and 10 runs were performed with faults injected with the same probability.

Experimental measures give here $\frac{nb\text{ iterations}}{nb\text{ contexts}} \approx 400$. There are indeed many more context systems compared to experiment 1 with most of them relevant to the solution computation (i.e. operators). This leads to the quantities:

- $q(p_c = 0.0001) = 30.72$
- $q_c = q \cdot \frac{35}{63} \approx 17$
- $q_s = q \cdot \frac{28}{63} \approx 14$

The estimation of damage is of about 31 characters damaged during the program execution, distributed as 17 characters (54.8%) of context systems and 14 characters (45.2%) of non context systems.

Figure 6.5 shows the results of such configuration tested over 10 runs.

![Figure 6.5: Experiment 2: GA progression without and with faults averaged over 10 runs with a program configuration using redundant systems](image)

Results show that the program performed well in its task in spite of the faults. However, if the execution had required to last longer (e.g. more difficult problem), or if more faults were to occur, the program could stop working before reaching its goal like in experiment 1. This hypothesis is verified in the following experiment.
Experiment 3: Raising the Fault Rate

This experiment uses the same system configuration as in the previous experiment but increasing the character-wise fault probability to \( p_c = 0.0005 \) giving the system corruption probability \( p_s = 0.3189 \). This system fault probability is comparable to a simultaneous erroneous state of a third of the systemic computer components.

- \( q(p_c = 0.0005) = 153.6 \)
- \( q_c = q \cdot \frac{35}{63} \approx q \cdot 56\% \approx 85 \)
- \( q_s = q \cdot \frac{28}{63} \approx q \cdot 44\% \approx 68 \)

The estimation of damage is of more than 153 characters damaged during the program execution, distributed as 85 characters (55.6%) of context systems and 68 characters (44.4%) of non context systems.

Figure 6.6 plots the obtained results showing that although using duplicated systems the program stops evolving before reaching its goal. When analysing the reasons of this program failure the insight is that solution transfer systems are the first not to fulfil their role anymore. Initialisers are indeed only required at the beginning, computation spaces are just encompassing systems so have no context nor data holding role, and solutions and operators (crossover or mutation) are more numerous than solution transfers. Analysing the results, looking at the systems memory state evolution through time, showed indeed that each program failure is due in the first place to the corruption of all transfer systems. If solution transfers were more numerous than operators for instance then solution evolution would be expected to stop working first. As soon as one program subtask (e.g. solution transfer, solution evolution, etc) is not fulfilled any more, the program stops working. In the case of the GA, the subtask in charge of transferring solutions to the universe is not executed anymore once all transfer systems are corrupted. Once such subtask is down, it does not matter what the others can do as the program requires all of them to work properly.

These experiments showed up to now that there is always a graceful degradation (solutions are evolved normally until evolution fails because of damage, but the solutions are not lost) but sooner or later the GA fails to work properly. The program failure point can be delayed by providing enough systems to survive for a while (e.g. as long as the program is required to run, see experiment 2) but it cannot be prevented if faults keep happening.
To slow down the degradation without adding too many systems, or even to avoid this failure point, the program can be repaired using a self-maintenance system. This is investigated in the next experiment.

**Experiment 4: Strong Fault Rate and Self-repair Systems**

This experiment repeats over 10 runs the same setup as experiment 3 but injecting in addition 7 self-maintenance systems. In this new configuration: \( \text{nb\_contexts} = 42 \) and \( \text{nb\_systems} = 70 \). The amount of self-maintenance systems thus represents 10% of the total amount of systems.

Experimental measures give here \( \frac{\text{nb\_iterations}}{\text{nb\_contexts}} \approx 535 \). This leads to the quantities:

- \( q(p_c = 0.0005) = 205.71 \)
- \( q_c = q \cdot \frac{42}{70} \approx q \cdot 60\% \approx 123 \)
- \( q_s = q \cdot \frac{28}{70} \approx q \cdot 40\% \approx 82 \)

The estimation of damage is of more than 205 characters damaged during the program execution, distributed as 123 characters (60%) of context systems and 82 characters (40%) of non context systems.
Note that $q_c$ increased with respect to $q$ as this configuration involved additional fallible context systems for self-maintenance. Figure 6.7 shows the program progression without faults, with faults and then with faults and self-maintenance.

![Graph showing program progression](image)

**Figure 6.7:** Experiment 4: GA progression with no fault, with faults and with faults and self-maintenance, all averaged over 10 runs with configurations using redundant systems and facing a strong fault probability

Results show that the program is working fine in spite of the high amount of faults (e.g. very unreliable hardware or very buggy software), and using a reasonable amount of systems dedicated to faults reparation.

**Discussion on Experiments Setup**

It should be noted that the more contexts there are in an SC program, the more iterations are required to make all contexts create an interaction once. Therefore, if the *laws of physics* system interacts once in a cycle, then the more systems there are, the less likely each individual system is to be damaged, but still the probability that something happens within the whole system is the same. Faults thus happen in a system depending on its usage. This bias is part of SC, just as any other paradigm can have inner biases due to their properties.

However, to remove this bias in the experiments, some dummy systems can be added to ensure that experiments with different needs of systems still have the same amount of systems (same amount of context systems and same amount of non-context systems).
To confirm this, experiment 3 was conducted again using 7 dummy context systems in order to have 42 context and 28 non-context systems like in experiment 4. This way the comparison between the two experiments was strictly unbiased. The results showed that the program running with faults performed on average only slightly better than in the dummy-less version, confirming that the bias had no significant impact on the overall outcome of the experiment.

6.3.2 Analysis

Experiment 1 demonstrated the fault-tolerance and crash-proof ability of systemic computing. It also showed the limitations regarding fault-tolerance on a program implementation not using redundant systems. Experiment 2 then demonstrated the positive impact of redundancy on fault-tolerance. To highlight the limits of any redundant configuration, experiment 3 showed by inflicting severe damage than even a significantly redundant configuration would stop working properly at some point. Finally experiment 4 demonstrated the self-repair ability of SC programs. This involved using self-maintenance context systems between similar systems in order to regenerate their broken parts as the program is undergoing constant severe damage.

The fault-tolerance ability has therefore been successfully demonstrated. The contribution to the evidence list given in Section 3.8 (page 129) is now discussed.

Improved Naturalness

Natural systems are built upon a distributed basis (e.g. organisms made of cells, molecules made of atoms) where entities interact locally (see Subsection 2.2.5, page 48). Such architecture can allow biological systems to cope with and recover from damage by exploiting the redundancy of constituting entities (e.g. DNA and cell replication). While conventional approaches to fault-tolerance (see Subsection 2.4.2, page 62) suggest alternate code to handle several potential scenarios (e.g. try-catch, n-version programming, recovery blocks), SC provided native fault-tolerance by imposing a model organisation that involves distributed and independent interacting entities. Experiments showed that the greater the redundancy of systems, the greater the fault-tolerance ability. Self-maintenance was then obtained by exploiting the systems redundancy, similarly to natural systems. Also the nature of the program was irrelevant to its crash-proof, fault-tolerance and self-repair potential, making the study relevant to any kind of program, not just GAs. Therefore as within natural systems, SC provides an approach to fault-tolerance present at the core of the models, as a part of their own nature.
Conciseness, Compactness and Readability

Conventional methods dealing with fault-tolerance introduce code overhead due to the alternate blocks of code and the fault-handling system that is provided (see Subsection 2.4.2, page 62). The SC approach integrates fault-tolerance at the core of a program with no code overhead, by using system replication. The self-repair ability was added using one kind of context system injected in the model. Self-maintenance systems can repair each other so no meta level of self-repair is required to maintain the self-repair ability. Both fault-tolerance and self-repair abilities are usable for any form of program, making the solution generic and minimal.

Properties Exploitation Ability

Systemic computing, by being a continuous process, cannot crash in the conventional sense. Any partial damage made to a program cannot be lethal for the computing process which may nevertheless stop working properly. As a result, SC programs embed fault-tolerance. However the degree of fault-tolerance achievement ranges from a program quickly losing its functionality to a program that keeps working properly in spite of repeated damage. The ability to cope with damage was obtained using systems redundancy. This redundancy exploits local and distributed computation such that a broken system can be replaced by another one handling the same role in the program. Self-repair was then obtained by exploiting further these properties and using healthy system parts to replace damaged ones within similar systems. The property exploitation scheme is summarised in Figure 6.8. It is also noteworthy that the self-maintenance systems, using healthy parts of systems to fix damaged ones on others, enable the program to maintain itself in a stable state and as such enable a homoeostatic behaviour.

6.4 Summary

This chapter demonstrated how nature-inspired programming using SC can natively provide fault-tolerant behaviour based on crash-proof computing and improve this ability with easily integrated self-maintenance to a program with minimal software conception overhead. These properties were obtained by exploiting the continuous, distributed and local properties of SC.

The continuity of systemic computing taking eligible systems, if any, for computation makes SC by definition crash-proof and hence fault-tolerant. A high level of fault-tolerance is then achieved by duplicating system instances to ensure that no system is left alone to fulfil a crucial task. Finally self-maintenance is implemented by introducing
Chapter 6 Fault-tolerance within a Genetic Algorithm

Figure 6.8: Properties exploitation diagram for an SC program (here a genetic algorithm) able to demonstrate fault-tolerance with self-repair. Continuous, distributed and independent computation enables crash-proofness. Distributed local computation enables fault-tolerance as the failure of some components (i.e. systems) can be compensated by other components with the same function. The program being crash-proof and able to demonstrate a tolerance to a high level of faults can then be made self-repair by injecting redundant self-maintenance systems.

Continuous computation

If possible, pick up context

If possible, compute

Continuous selection of potentially available context and interacting systems

Distributed computation

Solution

Solution

Initialiser

Initialiser

Distribution of duplicated agents holding various functions in the model

Solution Transfer

Solution Transfer

Genetic Operator

Genetic Operator

Computation Space

Computation Space

Self Maintenance

Self Maintenance

Exploitation of local information per computation

Local interactions

Solution

Solution

Solution

Solution

Genetic Operator

Self Maintenance

Crash-proof Fault tolerant process able to Self-repair

Computation is continuous disregarding what systems fail. A failure of all systems would lead to no computation, not to a crash.

The failure of some systems is compensated by others of the same type, enabling fault tolerance.

Redundancy also enables to copy healthy parts of systems to repair broken parts of other similar ones.

= Failing system
a new kind of context system using existing systems to repair each other. The fault-tolerant self-maintaining genetic algorithm used in the last experiment showed that SC can be a crash-resistant computer able to run fault-tolerant self-maintaining programs in spite of a high probability of fault occurrence and allocating only 10% of its resource to maintenance. Such SC program being able to maintain itself stable is thus also performing a form of homoeostatic behaviour.

With this method, fault detection and fault correction are done automatically and are fully integrated into the core of the program. The fault detection mechanism is independent from the kind of systems being repaired and can therefore be used in any SC program. Similar to a biological organism, this process is part of the whole and just as any other constituent is a regular and autonomous running task.

The next chapter exploits the asynchrony and local knowledge inherent to SC in order to create flexible neural structures.
Chapter 7

Flexibility within Artificial Neural Structures

7.1 Introduction

This chapter investigates the impact of local knowledge and asynchronous computation within SC. To investigate the benefits of built-in asynchronous computation and local knowledge, this chapter focuses on artificial neural networks as these properties are significant natural properties of biological neural networks. More specifically neural networks are suitable to investigate these properties as:

- Neurons are organised to create a whole (the network) that solves problems.
- Neurons are computing locally, yet the result is global.
- Neurons are independent (in timing and internal knowledge).

Local knowledge and asynchrony do not suit conventional computer architectures (see Subsection 2.2.3, page 41) and classical neural network models often employ global algorithms, constraining networks’ structure and taking networks away from their original concept (see Subsection 2.4.3, page 66). Real biological neural networks however imply a more flexible model without the structural limitations imposed by conventional approaches.

The use of SC requires the use of local knowledge and asynchronous computation. This work presents a bio-inspired model of artificial neural networks, focussing on agent interactions, and shows that exploiting these built-in properties of SC, which come for free, improves naturalness by enabling neural structure flexibility without reducing
performance. To assess the model, several neural structures are tested against various problems.

The work presented in this chapter was published in [Le Martelot and Bentley, 2009a, Le Martelot et al., 2007b].

7.2 Artificial Neural Network Model

7.2.1 Systemic Analysis

Modelling a neural network keeping its natural characteristics should involve the same entities that form a real one: neurons, their inner mechanism and their communication mechanism. These mechanisms could be modelled a priori at several levels. One model could represent the interaction of neurons using synapses to make the link between axon and dendrites. Another one could involve pre-synapse, post-synapse, protein exchange, protein transfer, etc. The chosen level of abstraction for this study is the neuron. There is hence no need to explicitly represent protein interactions. A neuron receives inputs from its dendrites that are processed in the soma; the resulting signal is then sent through the axon [Kandel et al., 1991]. Axon signals are weighted and transmitted to further neurons through synapses that communicate with their dendrites. The signal will thus be a value transmitted across the network rather than many molecular and electrical entities.

The synapse which transfers signals from axon to dendrites can be chosen as a context of interaction between neurons. However, neurons interaction do not provide information regarding the signal flow direction. This flow is by definition directional from axons to dendrites. Therefore the model should have the more precise notions of axons and dendrites to specify the signal direction. Dendrites can be modelled as one system representing the dendritic tree rather than one system per dendrite which would add unnecessary complexity to the model. A synapse connecting an axon with a dendrites system, each systems triplet belongs to the scope of a particular synaptic connection. This is given in the following calculus expression and summarised in Figure 7.1.

(Axon Synaptic_Connection Dendrites)

Synaptic_Connection ( Axon Synapse Dendrites )

Axon[signal] \rightarrow Synapse \rightarrow \{ Dendrites[signal_{k,t-1}] \rightarrow Axon[signal_t] Dendrites[signal_{k,t}] \}

Two types of synapses could be considered here: excitatory and inhibitory synapses [Kandel et al., 1991]; not to mention that synapses can be electrical or chemical [Kandel...
et al., 1991], but this is not explicitly modelled here. For modelling simplicity and so as not to introduce inconsistencies both excitatory and inhibitory excitations are allowed within one synapse. This is modelled by a weight taken within [-1; 1]. A positive weight simulates an excitatory synapse and a negative weight an inhibitory one.

To model the signal processing between dendrites and axon inside a neuron, the ionic transmissions in the membrane and the membrane can be considered as a whole and the membrane can thus be defined as context of interaction between dendrites and axon. A membrane also owns a threshold of signal activation as a real value also taken within [-1; 1]. To keep neuronal integrity, scopes are used to group what is part of a neuron, of the outside or of both. All the inherent neuron interactions happen within its soma (a membrane context system is located within a soma as the membrane belongs to the very soma). A neuron is therefore represented as dendrites, a soma, a membrane and an axon. However, dendrites and axons also belong to the outside (they are exposed to the outside of the soma) as their role is to receive and transmit signals from or to other neurons. Therefore, neurons can be modelled as described in the following expression and shown in Figure 7.2.

(Dendrites Soma Axon)

Soma (Dendrites Membrane Axon)

Dendrites[signal_{k,t}] - Membrane - { Axon[signal_{k-1}] \rightarrow Dendrites[signal_{k,t}] } Axon[signal_t]

Neurons belong to a neural network, therefore it is sensible to encompass them in a network system itself contained in the systemic universe. The universe encloses everything within the program and is also used as the interface between the program and the user. However, the network inputs and outputs as well as the data transfer between them and the universe are still to be defined. A real brain receives axons from neurons located outside, like light-receptive cells in the retina, and sends signals also outside, like to muscles. Thus, axons can naturally also play the role of network inputs and outputs. Then
Figure 7.2: SC model of a neuron showing the dendrites-axon interaction in the context of a membrane and within a soma

Input transfer and output transfer context systems transfer data between the universe and the input and output axons, as given with the following expressions:

\[
\text{Universe}[\text{in\_signal}_{k,t}] \rightarrow \text{Input\_Transfer} \rightarrow \{ \text{Axon}[k,\text{signal}_{t-1}] \rightarrow \text{Universe}[\text{in\_signal}_{k,t}] \text{Axon}[k,\text{signal}_{t}] \}
\]

\[
\text{Axon}[\text{output,signal}_{t}] \rightarrow \text{Output\_Transfer} \rightarrow \{ \text{Universe}[\text{out\_signal}_{k,t-1}] \rightarrow \text{Axon}[\text{output,signal}_{t}] \text{Universe}[\text{out\_signal}_{k,t}] \}
\]

Figure 7.3 summarises these concepts with a single neuron SC neural network.

Figure 7.3: SC artificial neural network made of 1 neuron and taking 2 inputs. The model implies external sensory cells/neurons which are not modelled except for their axons.

The calculus expressions for the hierarchy of such model with \( k \) inputs and one output are given below:
Chapter 7 Flexibility within Artificial Neural Structures

Universe(Input_Transfer_1 \cdots \text{Input\_Transfer}_k \text{Axon}_1 \cdots \text{Axon}_k \text{Axon\_output} \text{Output\_Transfer} \text{Network})

Network (Axon_1 \cdots \text{Axon}_k \text{Synaptic\_Connection}_1 \cdots \text{Synaptic\_Connection}_k \text{Dendrites Axon\_output})

Synaptic\_Connection_i (Axon, Synapse, Dendrites), with \( i \in [1, k] \)

Soma (Dendrites Membrane Axon\_output)

So far, this model can organise interactions disregarding the physical location of the neurons. Nonetheless, the notion of neuron neighbourhood can be easily handled using scopes. An area system can encompass neurons and neurons can belong to several areas. This partition and sharing of neurons would thus create neighbourhoods in the network. Note that the physical neighbourhood is defined by relationships between systems rather than by physical coordinates. Figure 7.4 shows a more complex network with areas.

\[ \text{Figure 7.4: SC artificial neural network with 4 inputs, 1 output and 3 areas sharing neurons. Each area defines a neighbourhood for the neurons inside.} \]
This network partitioning into areas using scopes also offers potential interest for future work. Some more interaction possibilities could then be added, injecting new context systems in specific areas, thus giving one a different potential and behaviour from another. In addition, from a biological modelling point of view, partitioning the network into areas is of relevance [Kandel et al., 1991].

### 7.2.2 Rules

The organisation of neurons is based on observations taken from biological studies [Kandel et al., 1991]. However, knowing the organisation does not explain the inner behaviour of the entities involved. Unfortunately, this is not well understood yet how everything happens at this stage. This implies using methods that may or may not be biologically plausible.

In the field of artificial neural networks, gradient backpropagation (BP) [Tang et al., 2007] is a method commonly employed as it can solve non-linear problems. Classical BP is often described as a global algorithm relying on some precise network structure. Indeed it constrains the network to be layered and feed-forward; therefore no change in the neurons organisation breaking this requirement can be made. Recurrent BP was introduced to overcome one of these constraints and cope with backward connections [Tang et al., 2007]. Other more biologically plausible techniques, like contrastive Hebbian learning for deterministic networks [Hinton, 1990, Peterson and Anderson, 1987], generalised recirculation [O'Reilley, 1996], or spiking neurons networks [Maass, 1997] were introduced and showed successful results. Still, these approaches all define global algorithms, coping with various specific network structures, giving neurons more and more realistic computational abilities, but do not give the neuron entity the ability to be autonomous (i.e. inner data processing) in whatever situation (i.e. disregarding the position in the structure). Such natural flexibility is, from a natural modelling point of view, what is desirable and missing in approaches using conventional computation. Therefore similarly to biological neural networks, the SC model should be flexible and made of independent neurons that can wire together without any structure constraints. As no learning rule seems suitable as it is for this model, the choice was to keep the principle of backpropagation but adapting it to be an asynchronous local-rule based principle.

BP relies on the concept of layers to group independent neurons together, which provides an easy control of the flow of information from layer to layer and therefore suits a global and serial algorithm. In the SC paradigm, the very same principle can be used without
any structure hypothesis by defining the information flow process locally. Equations 7.1, 7.2 and 7.3 give the backpropagation rules with a momentum factor:

\begin{align}
x_i &= g(h_i) = g(\sum_{k \in K_i} w_{ik} \cdot x_k) \tag{7.1} \\
\Delta w_{ik}(t) &= \lambda \cdot e_i \cdot x_k + \alpha \cdot \Delta w_{ik}(t-1), \forall k \in K_i \tag{7.2} \\
e_i &= g'(h_i) \cdot \sum_{j \in J_i} w_{ji} \cdot e_j = \sum_{j \in J_i} (g'(h_i) \cdot w_{ji} \cdot e_j) \tag{7.3}
\end{align}

with

- \(i\): the neuron for which a value is being processed,
- \(x_n\): the signal output of a neuron \(n\),
- \(g\): the transfer function,
- \(h_n\): the weighted input sum of a neuron \(n\),
- \(K_n\): the set of neurons firing into a neuron \(n\),
- \(w_{mn}\): the weight of the connection from a neuron \(n\) to a neuron \(m\),
- \(w_{mn}(t)\): the weight of the connection from a neuron \(n\) to a neuron \(m\) at a time \(t\),
- \(\Delta w_{mn}\): the variation of the weight from a neuron \(n\) to a neuron \(m\),
- \(\lambda\): the learning rate,
- \(\alpha\): the momentum term,
- \(e_n\): backpropagated error value of a neuron \(n\),
- \(g'\): the transfer function’s gradient,
- \(J_n\): the set of neurons a neuron \(n\) is firing into.

The mathematical principles of the rules can be kept. However their implementation needs to be local. Equation 7.3 shows that the error can be written as a sum. It can therefore be performed by a sequence of independent computations as long as their common term \(g'(h_i)\) remains constant during computations. Figure 7.5 shows where the values to be memorised can be stored in the systems and Figure 7.6 provides a flowchart of the error backpropagation and delta weight update.
Figure 7.5: SC diagram of a neuron with 3 inputs showing where data to be memorised can be stored in the systems. $x$ is a signal output from an axon, $h$ the weighted sum, $s$ a dendrite input signal value, $e$ an error value, $w$ a weight value and $\theta$ a threshold value. Axons hold a signal value to pass over through a synapse, and an error value to backpropagate. Synapses hold their weight, weight variation and last weight variation before update (used for momentum). Dendrites hold as many signals as they receive inputs (here $n$ would be 3). They also hold as many error data values as they have input connections so that each synapse manages its own data slot for both signal and error. Membranes hold their threshold, threshold variation and last threshold variation before update (used for momentum).

Each neuron keeps a current error value where further neurons can add their backpropagated error contribution. To avoid reusing twice any information, a neuron resets to zero its error value as soon as it is backpropagated. To assert the constancy of the terms $g'(h_i)$, the weights are updated only at the end of the computation (like in a classical BP) when a new sample is presented.

Figure 7.7 gives an example of the algorithm over a few steps illustrating the error backpropagation. The learning rate $\lambda$ is set to 0.5 and the activation function is set to the identity function for simplicity. No momentum factor is used. Note that neurons 4 and 5, and then neurons 1 and 2 are processed independently and not necessarily one after the other (or at the same time in a pure parallel environment). The computations
Chapter 7 Flexibility within Artificial Neural Structures

Figure 7.6: Flowchart of the error backpropagation and delta weight update. Neuron \( k \) receives error values from ahead neurons \( i \) and \( j \) and backpropagates error values to neurons \( m \) and \( n \). Error is transmitted between dendrites (De) and axons (Ax) by the context systems membrane (Me) and synapse (Sy). A synapse updates its weight variation \( \Delta w \), a membrane its threshold variation \( \Delta th \), \( \lambda \) is the learning rate and \( e \) is the error. The momentum term is not shown for simplicity and readability.
Chapter 7 Flexibility within Artificial Neural Structures

(a) Neurons 4 and 5 backpropagate an error value to the dendrites.

(b) Synapses compute the weights variations and backpropagate the error to the axon.

(c) The membrane computes the threshold variation and sends the error to the dendrites.

(d) Synapses compute the weights variations and backpropagate the error to the axons.

(e) When required, update all weights and thresholds and reset all variations.

Figure 7.7: Error backpropagation example using $\lambda = 0.5$ and $g(x) = x$. 
of neurons 4 and 5 and neurons 1 and 2 are shown together solely as it simplifies the illustration. Another scenario could for instance have neuron 4 backpropagating its error up to neurons 1 and 2, and then only have neuron 5 backpropagating its error to neurons 1 and 2. The total backpropagated error to neurons 1 and 2 would in the end be the same.

Neurons and synapses are therefore autonomous and responsible for their own local data. These local rules imply that no global algorithm is defined and therefore no constraint is imposed on the network’s structure. This model can be used to design feed-forward networks as well as recurrent networks as shown in Figure 7.8. The biological plausibility comes in this work from the autonomy and organisation of the systems, caused by the natural characteristics of asynchrony and local knowledge built into SC, leading to an emerging global behaviour, like the global learning. The model could potentially use any other kind of neuron (any neuron generation [Maass, 1997], i.e. different signal processing and learning methods) while keeping the very same organisation. Because the model’s structure imposes a neuron-based computation no global constraint can arise and limit the neural structure.

Note that a stabilised network could easily have its weak synapses trimmed by the injection of a new context system, programmed for instance to kill settled redundant synapses. This illustrates how the model could be improved by easy addition of new systems rather than requiring modifications of code at its core.

7.3 Experiments and Analysis

7.3.1 Experiments

To demonstrate the flexibility enabled within this model, three sets of experiments were conducted involving various structures each tested against an appropriate problem. Experiments were conducted using the SCANN application developed for this work and consisting of the virtual machine and a graphical user interface designed for artificial neural network models. The application is shown in Figure 7.9 and presented in Appendix A. Also, on-line (incremental) learning is favoured to batch learning for biological relevance reasons (natural computation is continuous, conventional computation is batch).
Figure 7.8: (a) A feed forward SC artificial neural network. (b) Same network with a recursive synapse. The program is initially the same but then topped up with one more synapse.
Experiment 1: XOR

Problem

The XOR problem is a common example of non-linearly separable problems, problems simple perceptrons cannot solve. Usually, backpropagation feed-forward multi-layer perceptrons are used and a common solution involves 2 hidden neurons and 1 output neuron [Yanling et al., 2002], as shown in Figure 7.10(a). However, the XOR problem becomes a linearly separable problem when adding a third input, doing for instance the AND of the 2 others [Yanling et al., 2002]. A 2 neurons network with 1 hidden neuron as shown in Figure 7.10(b) is thus enough to solve the XOR problem [Yanling et al., 2002].

(a) 3 neurons network  
(b) 2 neurons network

Figure 7.10: Two networks for the XOR problem
Chapter 7 Flexibility within Artificial Neural Structures

Setup

The two structures from Figure 7.10 can be simply created with the SC model by just assembling neurons together (i.e. adding the appropriate dendrites, soma and axon systems into the environment). In the experiments, both network structures were created and their performances were evaluated with the new local backpropagation rules.

The networks used a hyperbolic tangent sigmoid transfer function:

\[ \text{sig}(x) = \frac{2}{1 + e^{-2x}} - 1 \]

on all nodes. However, it sets the error on the output node using the gradient of the identity function (i.e. \( g'(h) = 1 \)) because the gradient of a sigmoid function on the last node would provide a poor error estimation. If the input signal on the node is around the worst value (i.e. 1 when the correct value should be -1) a sigmoid function gradient provides a low penalty (\( g'(h) \approx 0 \)) compared to the penalty a neutral value in input would receive (\( g'(h) >> 0 \)). The learning rate \( \lambda \) is set to 0.5 with no momentum.

The training set contains the four truth table expressions of the XOR operator. Samples are presented to the networks randomly during an epoch. The binary input values are \(-1 \) for false and \(1 \) for true. The networks are allowed 100 epochs for learning. Each test is run 25 times.

Results

Table 7.1 summarises the results. As the taught values are \(-1 \) and \(1 \), an answer was considered correct if for each expected \(-1 \) on an output node a negative value was returned, and for each expected \(1 \) on an output node a positive value was returned (e.g. if expecting a \(1 \), a value of 0.5 would be interpreted as correct, incorrect values would be any value lower than or equal to 0).

Table 7.1: XOR experiments results giving for each network implementation from Figure 7.10 the percentage of success (perfect truth table) over 25 runs in solving the problem.

<table>
<thead>
<tr>
<th>S.C. from 7.10(a)</th>
<th>S.C. from 7.10(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>84%</td>
<td>100%</td>
</tr>
</tbody>
</table>

The results show that both network show good performance in solving the problem. However, the network from 7.10(b) performs significantly better than the network from 7.10(a). The failures of the network from 7.10(a) are presumably due to some unfortunate initial values setting the network in a steady state from which it is difficult to move. Such network is indeed symmetrical and the change in value from one neuron on the input layer can potentially be cancelled by an opposite change in the other neuron.
on the input layer. The network from 7.10(b) however does not have this architecture issue and is using the minimum number of neurons (i.e. 2 in the XOR problem). As a result 100% of the network succeeded in settling in a state that provides a perfect truth table. Such observation suggests that, at least in some cases, a minimal neural structure is better than a structure with additional neurons.

**Experiment 2: Iris data**

**Setup**

The second experiment assesses another structure for data classification. It uses the well-known *iris plant* data set [Fisher and Marshall, 1988]. The data set contains three classes of fifty instances each, where each class refers to a type of iris plant (*Setosa*, *Versicolour* and *Virginica*). The first class is linearly separable from the other two; the latter are not linearly separable from each other. The data was normalised and centred before being presented to a network.

The network used has 4 inputs, 2 hidden neurons and 3 output neurons, as shown in Figure 7.11. The transfer functions are set as in experiment 1 (all sigmoid functions except on the last node, which is sigmoid with a linear gradient). The learning rate $\lambda$ and the momentum rate $\alpha$ are respectively set to 0.1 and 0.75. The maximum number of epochs was set to 50.

**Results**

Table 7.2 shows the classification results over 10 runs taking half of the samples (75) for learning and the other half (75) for testing. The aim is to assess the classification ability and the results clearly show that the SC model can perform well in such classification task.
Table 7.2: Average percentage of success over 10 runs of the 3 iris data sets and total percentage of success (L and T respectively stand for the learning and the testing sets).

<table>
<thead>
<tr>
<th></th>
<th>Setosa</th>
<th>Versicolour</th>
<th>Virginica</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L</td>
<td>T</td>
<td>L</td>
<td>T</td>
</tr>
<tr>
<td>S.C.</td>
<td>100</td>
<td>100</td>
<td>93.2</td>
<td>94</td>
</tr>
</tbody>
</table>

Experiment 3: Recursive network

Setup

The SC model can also handle recursive connections. One example of such a network is a simple signal amplifier (designed in this work and defined as a neuron using its signal to reinforce itself). In this third experiment, two networks shown in Figure 7.12 are created and learn a small amount of two sets to classify (Setosa and Versicolour, linearly separable). Each set consists of 50 samples and 5 of them only are given for each set to the networks to learn within 3 epochs. The networks are then tested with the rest of the sets’ samples. The learning rate $\lambda$ is 0.25 and no momentum factor is used.

Figure 7.12: A simple perceptron and a perceptron with a recurrent connection from its output to one of its inputs

Results

Table 7.3 provides the results. A network that had been perfectly trained would answer 1 or -1 (the taught values). The standard deviation of the response within one set would then be 0. Therefore the closer the response to the taught value (i.e. 1 or -1) the clearer the classification (a value of 0 meaning “not classified”). The smaller the standard deviation, the better a class of samples is identified.

The results show that the response is always stronger using the amplified perceptron than using the regular one while providing a lower standard deviation. Here learning has
Table 7.3: Average classification (i.e. network’s response) value over 10 runs for the Setosa and Versicolour (respectively taught at 1 and -1) sets with a simple perceptron and an amplified perceptron. *Average* gives the network’s response. *Std.Dev.* gives the standard deviation over the samples of each set.

<table>
<thead>
<tr>
<th></th>
<th>SC Perceptron</th>
<th></th>
<th>SC Amplified perceptron</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Std. Dev.</td>
<td>Average</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Learning</td>
<td>Setosa</td>
<td>0.8982</td>
<td>0.1415</td>
<td>0.9370</td>
</tr>
<tr>
<td></td>
<td>Versicolour</td>
<td>-0.8076</td>
<td>0.1314</td>
<td>-0.8645</td>
</tr>
<tr>
<td>Testing</td>
<td>Setosa</td>
<td>0.8769</td>
<td>0.0435</td>
<td>0.9264</td>
</tr>
<tr>
<td></td>
<td>Versicolour</td>
<td>-0.6092</td>
<td>0.2578</td>
<td>-0.6082</td>
</tr>
</tbody>
</table>

been performed on very few epochs and samples; the network should thus be insensitive to noise. The signal reinforcement allows a strong response in a short learning time without requiring a steeper sigmoid function.

Note that this is however a simple example designed to show the possibility of using recurrent connections within the neural network model. Here the learning method was indeed an adaptation for distributed local computation of the classical backpropagation method which is not initially intended for recursive networks. To fully exploit recurrence in more complex networks the learning method may need to be adapted, another method could be chosen, or some different kind of neurons could be added. Any learning method can potentially be implemented in the neural network model provided it relies on local computation only.

**Discussion on Potential Evolutions**

The aim of this work was to show that the SC neural network model has the potential for flexible neural structures. However, in addition, the model implementation was designed to cope with additional connections or connection loss. If a model was to grow or modify its connections, the dendrites could add or remove slots (holding input signals and errors, as shown in Figure 7.5) on the fly. Recent work investigated neural structure’s development and adaptation [Khan et al., 2007, 2008] using Cartesian genetic programming [Miller and Thomson, 2000]. In this approach genetic material encodes seven neural components that control the behaviour of the network at the neuron level (soma, dendrites, axon). Tested within an agent-based system experiments showed that these networks could enable a good understanding of environment’s signal, intelligent reaction to such stimuli and a form of memory. Results also showed that development was an important part of the learning process. These networks have also successfully been applied to the task of learning to play checkers [Khan and Miller, 2009]. In this respect, further work with the SC model could therefore investigate adaptive approaches where the structure of the network is modified on-line. The regulation of the growth
of new neurons could be handled similarly to the aforementioned work but also for instance aided by an artificial metabolism (see Chapter 8) that would manage the resource available (in SC, systems are transformed but created out of nothing or destroyed).

7.3.2 Analysis

These sets of experiments showed that the neural network model presented here could be used to create various structures such as non-layered and recursive. The work also explained how the error backpropagation technique could be adapted to such model.

Experiment 1 showed that a seemingly simple problem like learning the XOR truth table could be solved easily by a minimal non-layered network whereas constrained layered architecture would be less accurate. Experiment 2 showed that the network can perform accurate classification in a bigger non-linear classification problem using the well-known iris plant data set. Experiment 3 finally showed that recurrent connections can be added and provide better response without requiring a longer learning or a steeper activation function. Overall experiments showed that the accuracy of the method could compete with conventional backpropagation implementations.

Improved Naturalness

One reason for using SC is to move beyond simply attempting to mimic the functional behaviour of natural systems through global algorithmic approximations, and instead duplicate as much as is feasible the functional behaviour through mirroring the underlying systems, organisations and local interactions. In this respect the SC artificial neural network model involves dendrites receiving inputs processed within the membrane and sending an output signal through axons. These axons connect to further dendrites through synapses. Areas can be defined using the concept of scopes in order to set regions of the network. Therefore the SC model is significantly closer to biological neural networks than conventional approaches. As a result the model is flexible and neurons can be wired together with no structure restriction: synapses can potentially be added or removed, no layer architecture is required and recursive connections can be used.

Conciseness, Compactness and Readability

The nature of SC computation suggested a model focusing on local entities rather than global concepts such as neuron layers. The resulting model involves dendrites, membranes, axons and synapses that can be arranged in any manner, organised in areas, as illustrated in Figure 7.4. This model is more detailed than conventional approaches and encompasses information regarding the flow of information by using the notions of
dendrites and axons. Therefore while such details yield a more complicated representation compared to the conventional graph-like nodes and connections between them, they provide a more precise model containing a greater amount of information regarding the underlying process. Also the emphasis on relationships between systems results in a clear and readable model, as opposed to less adapted methods such as the stochastic π-calculus (see Subsection 3.4.3, page 120).

Properties Exploitation Ability

In this model, distributed asynchronous and local computation is used to create a neural network model without any global process. The model uses independent entities (e.g. dendrites, membrane, axon, synapse systems) that require signal processing and learning to be based upon local knowledge. Any signal processing and learning method implementation can potentially be used in the model. As a result the neurons can be assembled in any manner and the behaviour of the network emerges solely from local interactions in the structure. By imposing such organisation the SC neural network model is therefore flexible and does not impose any specific structure limitation, as illustrated in Figure 7.13.

7.4 Conclusion

This chapter showed how the intrinsic SC properties of local knowledge and asynchrony naturally provide more flexibility for artificial neural network structures. Experiments demonstrated that this flexibility is obtained without reducing accuracy. The example implementation contrasts significantly with classical approaches where data and algorithm are interdependent separate parts making network implementations more rigid and less biologically plausible. The SC implementation gave full autonomy to neurons, and is compatible with any neuron model (first, second, third generation [Maass, 1997]).

The next chapter investigates another bio-inspired model, an artificial organism, in order to study self-organisation and homoeostasis.
Figure 7.13: Properties exploitation diagram for an artificial neural network implemented in SC. Signal is processed locally within the context of a synapse or a membrane independently from other interactions taking place. The combination of asynchronous distributed and local computation leads in the implementation of an ANN to a flexible neural structure unconstrained by learning rules.
Chapter 8

Self-organisation and Homoeostasis within an Artificial Organism

8.1 Introduction

Throughout the course of time, evolution allowed species to progressively change and adapt to their environment by modifying their gene pool [Darwin, 1859, Kutschera and Niklas, 2004]. However, the genetic material is just the basic information necessary to build an individual. From the state of embryo (multicellular diploid eukaryote) organisms develop towards more mature states (e.g., development of cellular tissue). Also, throughout their lives, organisms have to maintain themselves in a stable state that ensures a good health and which can be restored when a pathogen threatens and shakes this equilibrium (e.g., regrow damaged parts of a tissue). The process in charge of controlling and maintaining this balance is the metabolism [Berg et al., 2002]. Within development and metabolism, organisms thus demonstrate inner self-organisation and homoeostatic behaviour.

This chapter investigates the properties of self-organisation and homoeostasis. For this study, the model developed is a new kind of program, an artificial organism, a program with its own metabolism. The motivation for such a model comes from an increasingly popular view in the field of artificial immune systems (AIS) which holds that innate immunity (as enabled by non-adaptive cells such as dendritic cells) can play a significant role in maintaining immunity in computer systems [Aickelin and Greensmith, 2007]. Notions such as the danger theory suggest that normal self cells may provide signals when damaged, thus helping to encourage the response of immune cells in the right
areas of the tissue of an organism at the right time [Matzinger, 1994]. Previous work has investigated the development of an artificial tissue to serve this function, providing an interface between data and artificial immune systems, and performing preliminary data processing and clustering [Bentley et al., 2005]. Another motivation for this study is the exploitation of the SC rule stating that systems cannot be destroyed or created from nothing. As a result any model that involves the incremental growth of new systems must also involve some method for eating (converting environmental systems into new parts of itself). Likewise, any model that involves death (without an instant replacement as was used in the earlier GA models) must also involve the expulsion of waste systems. Thus this form of model may form the basis of many other similar kinds of model, necessitated by the physical realism of SC.

This work extends the previous work on tissue for AIS by focussing on the properties of self-organisation and homoeostasis. In contrast to previous implementations of tissue, which largely ignore the relationships between real organisms and their environments, this work deepens the biological analogy by modelling an artificial organism as a program with metabolism. The program does not only mimic some tissue features but also mimics many fundamental properties of living organisms: eating data as food and expelling waste, while growing tissue, and releasing danger signals [Matzinger, 1994] when its cells die in an abnormal way. To assess the self-organisation and homoeostasis abilities the implementation is tested by application to two standard machine learning sets. Tested against the breast cancer dataset [Wolberg et al., 1995], the organism shows excellent abilities to recognise anomalies in its diet. Self-organisation capabilities are then investigated using first an ideal dataset to illustrate how the organism is ideally behaving, and then using the UCI Wine [Forina et al., 1991] dataset to assess it in a real case study.

The work presented in this chapter has been published in [Le Martelot and Bentley, 2009a,b, Le Martelot et al., 2008b].

### 8.2 Additional Background

Although not commonly modelled, the notion of tissue is fundamental to immunity. The immune system within an organism defends the tissue of that organism. The concept of artificial tissue has been used for instance in the POEtic project, aiming at creating a hardware platform organised with a similar hierarchy as found in biological systems [Tempesti et al., 2002], and using reconfigurable circuits to simulate tissue growth [Thoma et al., 2004]. It has also been used in work that implemented an AIS
in a sensor network, the sensor nodes taking on the role of tissue cells [Wallenta et al., 2008].

In biology, tissue is a crucial part of the immune system and its importance was particularly highlighted by Polly Matzinger when introducing the danger model [Matzinger, 1994]. This view rejected the notion that the immune system differentiates self from non-self and suggested that it instead responds to cellular damage. It thus suggests that cells that die abnormally release signals which encourage immune cells to converge on that location and become more active.

This theory was adopted in [Bentley et al., 2005] to propose two ways of growing tissues, a network tissue growing algorithm and a swarm tissue growing algorithm, where damaged cells would release danger signals exploitable by an AIS. Tissue was defined as the interface between a problem to solve and the AIS. The two approaches were independent of the size of the dataset. They would take an input stream of data and cluster data based on similarities while using danger signals to signal potential anomalies.

Here a similar view is followed, but the work attempts to improve the tissue model and its potential advantages by implementing using systemic computation a tissue-growing program designed for AIS as a self-organised artificial organism demonstrating homoeostatic behaviour. The model is used for anomaly detection but also for data classification and analysis.

8.3 Artificial Organism Model

8.3.1 Systemic Analysis

In most artificial immune systems, the level of abstraction is the cell: few approaches require modelling of the internal organelles or genome of cells, and few require modelling of populations of organisms. Here the growth of tissue cells, the consumption of food (data items), the expulsion of waste and the emission of danger signals will be modelled. Thus an abstraction at the cellular level is appropriate, with systems being used to explicitly model each element.

The identification of appropriate low-level systems is aided by an analysis of interactions. The organism should be able to eat food from its environment, use this food to grow organs (clusters of cells) by creating new cells and expel waste into the environment.

To prevent being overloaded with systems, the waste can be recycled into new food (a simple ecosystem model). Food and waste could therefore be seen as different states of
the same system. (In SC, as seen in Section 1.2 on page 21 and Section 3.3 on page 112, one rule states that systems can be transformed but never created from nothing or destroyed.) Also, the food is what the organism takes from its environment to be able to grow. Therefore cells and all the necessary matter for the growth should also derive from the food systems.

The ecosystem between the organism and the environment can be represented as shown in Figure 8.1.

Looking within the organism, it takes food as input and this food must be sufficient to grow tissue. One simple way to model this is by using the approximation that the food is transformed into cells when absorbed by the organism. However, to enable cells to adhere to each other (rather than float free), cells need some sticky adhesion molecules. Here all these molecules do not need to explicitly be modelled but an adhesion surface is at least required to bind two or more cells together. As SC forbids the creation of systems from nothing, the adhesion surfaces must be obtained either from incoming food or from the cells themselves. In a biological organism each cell has a limited lifespan and thus dies at some point. It may then be consumed by macrophages or dendritic cells and its energy is partially recycled. In the model dead cells can thus be recycled to make adhesion surfaces. A growth process can now attach cells to each other by using adhesion surfaces to create tissue. To regulate this growth and introduce the notion of time, a decay process simulates the ageing of cells. When cells die, a split process splits them from the adhesion surfaces they are bound to.
So the organism eats new data, converts each data item into a new cell, and attempts to bind that cell to others, with cells made from similar data items binding to each other. Thus, a cell unable to bind to any group of cells reveals itself to be significantly different from them - more like the result of a cell infected by an abnormal pathogen. If this abnormal cell dies unbound, it can therefore be spotted as a potential anomaly. In that case, the death of the cell can entail that cell releasing a danger signal (i.e. the cell can be converted into a signal). This signal can then be used by an AIS algorithm which can be implemented through the addition of systems corresponding to immune cells.

The organism can also make use of a hunger parameter defining a maximum amount of alive cells it can contain at a time. This parameter can be stored in the organism system and the absorption context then only allows food absorption if the organism is hungry. This parameter can be useful to avoid having the organism growing too big and using too much memory/data at a time. A bad usage of memory could indeed to some extend slow down the computation process significantly.

The organism’s food to waste chain is therefore as shown in Figure 8.2.

![Figure 8.2: Food to Waste cycle within the organism: Food is absorbed, transformed into cells. When dying cells can be recycled into adhesion surfaces if they were part of a tissue or turned into a danger signal if they were single. Cells, adhesion surfaces and danger signals have a limited lifespan and decay over time (i.e. when they reach a certain age they die). When dying, cells also need a split process to detach them from the tissue they were part of.](image-url)

From this defined cycle, the interactions and systems in the model can be written in the SC calculus form as follows (also see Figure 8.3):

- `organism |- absorb -{ food } organism ( cell )` - absorbs food into a new cell.
- `cell |- growth -{ adhesion_surface } cell ( adhesion_surface )` - a cell grows into an adhesion surface.
- `cell ( adhesion_surface ) |- split ( cell adhesion_surface )` - a cell splits into a new adhesion surface.
- `organism ( cell ) |- cell_recycling organism ( adhesion_surface | danger_signal )` - a cell recycles into an adhesion surface or danger signal.
- `X[age] ( time ) |- decay X[age+1] ( time ), X = cell | adhesion_surface | danger_signal` - a cell decays over time.
- `organism ( X ) |- expel ( organism waste ), X = adhesion_surface | danger_signal` - an organism expels waste.
- `universe ( waste ) |- waste_recycling universe ( food[data] )` - waste is recycled back into food.
The absorb system models endocytosis (e.g. via cell receptors), the growth system models the organism’s genome, the decay models the aging (progression along the axis of time), the split system models a chemical breakdown between adhesion molecules and cell wall, the cell recycling models the phagocytes, the expel system models exocytosis, waste recycling systems model the ecosystem, the universe models the environment, the organism system models the boundary between tissue and environment, food systems model nutrients, cells model tissue cells, adhesion surfaces model adhesion molecules, danger signal systems model Matzinger’s danger signals, waste systems model cell waste (unused or unusable compounds), and the time system models the dimension of time.

The model is initialised in the following state:

\[
\text{universe} = (\text{organism} \ \text{absorption}_{1} \cdots \text{absorption}_{n_{a}} \ \text{food}_{1} \cdots \text{food}_{n_{f}})
\]

\[
\text{organism} = (\text{absorption}_{1} \cdots \text{absorption}_{n_{a}} \ \text{adhesion\_surface}_{1} \cdots \text{adhesion\_surface}_{n_{s}} \ \text{growth}_{1} \cdots \text{growth}_{n_{g}} \ \text{cell\_recycling}_{1} \cdots \text{cell\_recycling}_{n_{c}} \ \text{expelling}_{1} \cdots \text{expelling}_{n_{e}})
\]

\[
\text{data}_{i} = (\text{time} \ \text{decay}_{i}, \ \text{split}_{i}), \text{ with } i \in [1, n_{d}]
\]

where \(n_{w}, n_{a}, n_{f}, n_{s}, n_{g}, n_{c}\) and \(n_{e}\) are respectively the number of waste recycling, absorption, food, adhesion surface, growth, cell recycling and expelling systems.

Figure 8.3 summarises the organism’s organisation and shows the potential interactions. Food is absorbed and turned into cells. Cells can bind to others via adhesion surfaces depending on the data they carry. Each cell contains the dimension of time and a decay process. The decay process increases an age counter held in each cell. When the age reaches the cell’s lifespan, the cell is marked as dead. The split process can then detach the dead cell from the adhesion surfaces it was bound to. If the cell was previously bound to other cells (via adhesion surfaces) the cell is recycled (by the cell recycling system) into an adhesion surface whereas if the cell remained single it turns into a danger signal. An adhesion surface can last a minimum of the cell’s lifetime but would not die as long as it is bound to at least one cell (to avoid breaking clusters for non data related reason). However cells cannot bind to an adhesion surface once its lifetime is elapsed. Danger signals decay like cells. Once a danger signal or an adhesion surface is dead (and not connected to any cell in the case of an adhesion surface) it can be expelled out of the organism and turned into waste by the expel system. Waste can finally be recycled by the ecosystem (waste recycling system) into new food.

Note that waste recycling, absorption, cell recycling and expel systems should have the same amount of instances. Indeed, on average if one food system can be created at a time, then only one can be absorbed, then recycled and finally expelled at a time.
8.3.2 Data Organisation within the Artificial Organism

So far an organism has been modelled within SC. To use this organism to organise the data, the data processing method has to be defined.

To incorporate data into the organism’s metabolism, new data items are placed into food systems, where it is stored in the schemata. Data from an incoming stream can be introduced when recycling waste. (In this implementation the data is read from a file, but it could also be read on the universe receiving data from external sources.) The amount of waste recycling and absorption systems gives the data introduction rate (the more food can be absorbed at a time, the more data are introduced). The data are then absorbed into the organism and transformed to cells. When a growth interaction occurs between a cell and an adhesion surface, the two are bound based on their data similarity. Algorithm 8.1 describes in pseudo-code the binding method. For binding a
cell to an adhesion surface the adhesion surface is injected into the cell but remains also part of the organism so that more cells can bind to it.

**Algorithm 8.1** Pseudo-code for the growth context binding method. \( \tau \) is a given threshold. The distance function calculates the Euclidean distance of two vectors.

```
if adhesion surface not bound to anything then
    Bind cell and surface
    Surface data value ← Cell data value
else if distance(Cell data, Surface data) ≤ \( \tau \) then
    Bind cell to surface
    Surface data value ← Average(Surface data, Cell data)
end if
```

The measure chosen in this implementation to compare data is the Euclidean distance (as was used in [Bentley et al., 2005]). In the organism, cells bind together according to their values, and various clusters may emerge from this, thus reflecting the data distribution. If a cell is left single then it means it cannot bind anywhere and therefore holds data significantly different from the current most common data values contained within the organism. This cell is then turned into a danger signal holding the data that an AIS could use to develop antibodies.

### 8.3.3 Parameters Setup and Behavioural Analysis

**Mathematical Modelling**

Depending on how many instances of each system are provided in the organism, its behaviour can vary. For a given absorption rate, a longer cell’s lifespan would keep cells longer in the organism, therefore making it bigger. More growth systems would give a higher probability for a given cell and a given surface to interact. To understand better the implications of each parameter, the organism’s behaviour can be analysed in terms of probability. A good measure of the behavioural accuracy of the organism is for a given cell holding a datum from class \( i \) the probability \( p_{c_i} \) to be picked and then bound to other cells from class \( i \) via any surface it is eligible to bind to.

The parameters in the organism are:

- \( n_g \): amount of growth systems,
- \( c_l \): cells’ lifespan,
- \( d_{r} \): data introduction rate.

The aim is therefore to express the probability \( p_{c_i} \) with these parameters.
The probability for respectively a cell and an adhesion surface to be picked by a growth context are:

\[ p_{\text{pick}}(c) = \frac{1}{n_c} \quad \text{and} \quad p_{\text{pick}}(s) = \frac{1}{n_s} \]

with \( n_c \) the estimated amount of cells in the organism and \( n_s \) the estimated amount of adhesion surfaces.

The estimated amount of cells \( n_c \) depends on the cells’ lifespan and the data introduction rate:

\[ n_c = c_l \cdot d_r \]

Cells being recycled into adhesion surfaces, the amount of adhesion surfaces \( n_s \) is the amount of cells \( n_c \) less the amount of danger signals \( n_d \). However here adhesion surfaces are recycled only once all cells attached to it died. Assuming a new cell got attached at the end of the life of a surface, the amount of surfaces can reach

\[ n_s = n_c - n_d + n_c = 2 \cdot n_c - n_d \]

The expected amount of cells in the organism for a given class \( i \) of the dataset \( E(c_i) \) is given by:

\[ E(n_{c_i}) = p_{r_i} \cdot n_c = p_{r_i} \cdot c_l \cdot d_r \]

where \( p_{r_i} \) is the introduction probability of any sample of class \( i \) and \( \sum_{k=1}^{n_{cl}} p_{r_k} = 1 \). \( n_{cl} \) is the number of classes of data in the dataset.

If classes samples are equally introduced, then

\[ p_{r_i} = \frac{1}{n_{cl}} \]  \hspace{1cm} (8.1)

This will be referred to as event \( Eq. \)

Single adhesion surfaces bind to single cells disregarding their data content, if only new cells and new surfaces are considered, the expected amount of adhesion surfaces linked to cells of a given class \( i \) is \( E(n_{s_i}) \) given by:

\[ E_{\text{new}}(n_{s_i}) = n_s \cdot p_{r_i} \]

However, throughout a run \( E(n_{s_i}) \) can vary as the organism can also bind cells in an unwanted way (i.e. cells from different classes bound together). In such case a surface initially bound to one cell of class \( i \) could then also be bound to more cells of another class \( j \), making this surface mainly linked to class \( j \) data. Therefore Equation 8.2 becomes
Equation 8.3 throughout execution:

\[ E(n_s_i) = n_s \cdot p_r \cdot (1 - p_{lose(i)}) + \gamma_i \]  

(8.3)

where \( p_{lose(i)} \) is the probability that a surface initially linked to a cell of class \( i \) would then bind together more cells of another class, and \( \gamma_i \) is the amount of surfaces initially bound to another class that became bound to more cells of class \( i \).

To study the behaviour of the organism the ideal case where the organism binds cells accurately so that \( p_{lose(i)} = 0 \) and \( \gamma_i = 0 \) along the execution can be considered. This ideal event will be called \( Id \). Simplifications of further equations using this assumption are thus given using the conditional probability notation.

For a given class \( i \) the probability to pick a surface bound to cells of class \( i \) can be expressed as

\[ p_{s|i} = \frac{E(n_{s_i})}{n_s} \]

and

\[ p_{s|i,Id} = p_r \]

therefore for a given cell of class \( i \) the probability to be picked and tested against a surface bound to cells of class \( i \) is

\[ p_{s|c_i} = \frac{E(n_{s_i})}{n_c \cdot n_s} \]

and

\[ p_{s|c_i,Id} = \frac{p_r}{n_c} \]

The probability to be picked and tested during a cycle is then

\[ p_{cycle \ s_i|c_i} = 1 - \left( 1 - \frac{E(n_{s_i})}{n_c \cdot n_s} \right)^{n_g} \]

and

\[ p_{cycle \ s_i|c_i,Id} = 1 - \left( 1 - \frac{p_r}{n_c} \right)^{n_g} \]

Finally the probability \( p_{c_i} \) is therefore:

\[ p_{c_i} = 1 - \left( 1 - \frac{E(n_{s_i})}{n_c \cdot n_s} \right)^{n_g \cdot c_i} = 1 - \left( 1 - \frac{1}{c_i \cdot d_r} \cdot \frac{E(n_{s_i})}{n_s} \right)^{n_g \cdot c_i} \]  

(8.4)
Using our ideal behaviour assumption, this probability can be written as:

\[ p_{ci|Id} = 1 - \left( 1 - \frac{pr_i}{cil \cdot dr} \right)^{ng \cdot cil} \]  

(8.5)

If class samples are introduced with equal probability, then using Equation 8.1 Equation 8.5 becomes:

\[ p_{ci|Eq,Id} = 1 - \left( 1 - \frac{1}{ncil \cdot cil \cdot dr} \right)^{ng \cdot cil} \]  

(8.6)

These results are based on expected values of cells \((E(n_{ci}))\) or surfaces \((E(n_{si}))\) but ignore for instance the probability of having a lower amount of cells from a class than the expected value. For a given class \(i\) the probability of having no cell from this class at a given time in the organism is

\[ p_{no\_cell(i)} = (1 - pr_i)^{nc} = (1 - pr_i)^{cil \cdot dr} \]  

(8.7)

Also in the organism, cells bind according to data similarities. So far data of a same class were considered to be straightforward to identify but this identification is part of the process in the organism and its hardness depends on the dataset. So, calling \(p_{can\_bind(ci)}\) the probability for a cell of class \(i\) to have a cell (and a surface) from class \(i\) to bind to, the following can be stated:

\[ p_{can\_bind(ci)} = (1 - p_{no\_cell(i)}) \cdot \delta_i \]  

(8.8)

where \(\delta_i\) is the dataset compactness factor for class \(i\) taking values within the range \([0, 1]\) (i.e. towards one, dataset is compact, towards zero it is sparse).

**Parameters Impact Analysis**

\(ng\): Equation 8.4 shows that increasing the amount of growth systems increases \(p_{ci}\). The downside of this parameter is from a memory point of view, the higher \(ng\), the more systems required for the growth task.

\(dr\): Equation 8.5 shows that increasing the data introduction rate decreases \(p_{ci}\). However, this can be counterbalanced by increasing \(ng\). (A study of \(p_{ci}\) at the limit with both \(dr\) and \(ng\) set to infinity shows that \(p_{ci}\) converges to a constant value.) Therefore, a bigger amount of growth systems could be used to maintain a same \(p_{ci}\) at a higher introduction rate, but this would only be relevant if the rate was significantly increased. Finally, Equation 8.7 and Equation 8.8 also show that a higher data rate increases the probability for a cell to bind to another cell of its class.
Chapter 8 Self-organisation and Homoeostasis within an Artificial Organism

$c_l$: Studying the slope of Equation 8.5 shows that it continuously goes down. Investigating the limit towards infinity of $c_l$ gives:

$$\lim_{c_l \to \infty} p_{cl|I} = \lim_{c_l \to \infty} 1 - \left(1 - \frac{p_{cl}}{c_l \cdot d_r}\right)^{n_y \cdot c_l} = 1 - e^{-\frac{n_y \cdot p_{cl}}{d_r}}$$

Therefore as $c_l$ increases $p_{cl}$ goes down to the value $1 - e^{-\frac{n_y \cdot p_{cl}}{d_r}}$. However, from Equation 8.7 and Equation 8.8 it is clear that the higher $c_l$, the higher $p_{can\text{-}bind(c_l)}$, therefore the chosen value for $c_l$ cannot be taken as low as Equation 8.5 only would suggest. A trade-off has to be made.

Number of classes: Looking at Equation 8.6, it can be observed that $n_{cl}$ has an impact on $p_{cl}$. The greater $n_{cl}$, the lower $p_{cl}$. $n_{cl}$ is not a parameter, it is a property of the dataset. Therefore $n_{cl}$ is a value indicating the difficulty of the classification problem for the organism. Clearly, for a given organism, the more classes, the harder to classify the data.

8.4 Experiments and Analysis

To assess the self-organisation and homoeostasis achievement within the model and to validate its mathematical analysis, a series of experiments were performed using two standard UCI machine learning datasets and one ideal dataset. The organism shows self-organisation if it can grow tissue from a data flow organised so as to reflect the data distribution. The organism shows homoeostatic behaviour if once a mature state reached it maintains itself in a stable condition (i.e. where the tissues stop growing in size, some cells die while new cells replace them) and if it is able to cope with anomalies in the data flow (metabolism). First, organism growth and anomaly detection was investigated using the UCI breast cancer dataset [Wolberg et al., 1995], also comparing the results with previous work [Bentley et al., 2005] and looking at the state of the program over time. Self-organisation within the organism is then investigated further using first an ideal dataset to provide an illustration of an ideal behaviour, and then using the UCI Wine [Forina et al., 1991] dataset.

8.4.1 Organism’s Growth and Anomaly Detection

This set of experiments uses the breast cancer dataset [Wolberg et al., 1995], comprising 458 benign items (class 1) and 241 malignant items (class 2), each item being a vector of 9 real-valued numbers that were normalised to lie within $[0,1]$. The choice of this dataset is ideal to test the growth of the organism using a data flow while controlling
the ability to detect anomalies. Indeed the organism should grow and live using the benign data by clustering them together while detecting as anomaly the malignant data by emitting danger signals. Choosing this dataset also enables comparison of anomaly detection performance with similar previous models [Bentley et al., 2005].

A. Experimental parameters analysis

For a given dataset, various configurations of the organism may provide different results. The first set of experiments investigates what setups are particularly efficient for anomaly detection in this dataset.

Setup

17 experiments were run, involving as parameters:

- \( \tau \): the data comparison threshold,
- \( d_r \): the data absorption rate (i.e. amount of absorption systems),
- \( c_l \): the cell’s lifespan,
- \( n_g \): the amount of growth systems.

The data comparison threshold \( \tau \) is the distance limit that determines if two data items can bind together (distance lower than the threshold) or not. This parameter is therefore an essential criteria of the classification. The accuracy of the threshold depending on the dataset (for a vector of 9 values within \([0,1]\], the maximum Euclidean distance is 3), the threshold values were first taken as low as 0.2 and then increased by 0.1 as long as the accuracy was improved. (Note that 16 data items within the dataset have a missing attribute, in which case the attribute is exceptionally not involved in the calculation of the distance between data.)

The other three parameters \( d_r \), \( c_l \) and \( n_g \) were previously analysed in Subsection 8.3.3 and found to be impacting the classification results. The data introduction rate was first taken at the minimum value 1 and increased by doubling each time its value until 24 to evaluate its impact. Cell’s lifespan was arbitrarily set to 15 when varying the other parameters, and then varied up (20) and down (10, 5) to investigate its impact. Finally the amount of growth systems was arbitrarily set to 100 when varying the other parameters, and varied up (150) and down (50, 25, 10, 5) to investigate its impact.

All experiment settings involve a universe, an organism, a time system, an equal amount of waste recycling, absorption, cell recycling, expelling system which value depends on the experiment (given by the data introduction rate), 250 data systems in experiments
1 to 12 and respectively 500, 750, 1000, 1250 and 1750 in experiments 13, 14, 15, 16 and 17 (more data systems are made available for larger organisms simply to ensure that no shortage of systems can occur). Each data system contains a decay and a split system. The organism initially contains 5 adhesion surfaces. The hunger parameter was set to 200.

Each experiment was repeated 20 times. Each run consisted of 3000 iterations with randomly picked data presented each iteration. Class 1 is treated as the normal class of data and class 2 is treated as abnormal. One data item is introduced from class 2 on average for 25 data items of class 1 (these values are taken from [Bentley et al., 2005] to enable comparison), i.e. with a probability of \(\frac{1}{26}\).

Results

Table 8.1 shows the results of various tunings that were used. These results are computed by discarding the early computations (here the first 50 000 computations were discarded) during which the organism grows to an adult (stable) state and stopping the experiments when the flow of data ends (thus not permitting organism’s death from starvation). (Preliminary experiments assessed the number of computations required for the organism to reach its adult stable state, and to assess the consistency of the value. The exact value can depend for instance on the absorption rate or the food availability.) Accuracy is measured using the average percentage and standard deviation (over the 20 runs) of data from each class creating a danger signal. Class 1 being normal data, it should ideally generate no danger signal. On the contrary class 2 should ideally generate 100% of danger signals. Standard deviation is used to measure that the behaviour is consistent over the runs, a low standard deviation (as is the case in all experiments) indicating that all runs have similar results and therefore that the average is an accurate measure.

From the first four experiments it can be observed that the data similarity threshold (\(\tau\)) has a significant impact on the performance of the program. The bigger the threshold, the less benign items are left unbound but the more malignant cells can potentially bind somewhere. A threshold of 0.4 was then used for the next experiments.

Experiments 5, 6, 3 and 7 show that increasing the lifespan (\(c_l\)) lowers the anomaly detections of class 1 data whilst slightly increasing the one of class 2. Anomaly detection (false positive) for class 1 decreases less and less as \(c_l\) increases, as described in Equation 8.9. It also shows, as highlighted by Equation 8.7 and Equation 8.8, that a too short lifespan (e.g. \(c_l = 5\)) gives significantly less accurate results than a higher value (e.g. \(c_l = 10\)). For malignant data (class 2), staying longer in the organism can only increase the chances of eventually binding somewhere by mistake. The lifespan should thus be long enough for class 1 samples to bind correctly without being overly long.
Table 8.1: Percentage of the average and standard deviation (over the 20 runs) of data from each class from the breast cancer dataset creating a danger signal (false positive for class 1 and true positive for class 2) for various setups. Parameters are respectively in order: new data introduction rate per cycle \((d_r)\), data comparison threshold \((\tau)\), cell’s lifespan \((c_l)\), and amount of growth systems \((n_g)\). Experiments 1-4 investigate the effect of varying \(\tau\), experiments 5, 6, 3, 7 investigate the effects of varying lifespan, experiments 8-11, 3, 12 investigate changing the amount of growth systems, experiments 3, 13-17 investigate varying the data introduction rate.

<table>
<thead>
<tr>
<th>Exp</th>
<th>(d_r)</th>
<th>(\tau)</th>
<th>(c_l)</th>
<th>(n_g)</th>
<th>Class 1 mean</th>
<th>Class 1 stddev</th>
<th>Class 2 mean</th>
<th>Class 2 stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.2</td>
<td>15</td>
<td>100</td>
<td>22.70</td>
<td>1.27</td>
<td>99.87</td>
<td>0.39</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.3</td>
<td>15</td>
<td>100</td>
<td>11.54</td>
<td>0.76</td>
<td>99.63</td>
<td>0.69</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>7.56</td>
<td>0.51</td>
<td>98.72</td>
<td>1.01</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.5</td>
<td>15</td>
<td>100</td>
<td>5.39</td>
<td>0.67</td>
<td>96.19</td>
<td>1.69</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.4</td>
<td>5</td>
<td>100</td>
<td>9.16</td>
<td>1.11</td>
<td>99.20</td>
<td>1.17</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0.4</td>
<td>10</td>
<td>100</td>
<td>7.85</td>
<td>0.48</td>
<td>99.06</td>
<td>1.14</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0.4</td>
<td>20</td>
<td>100</td>
<td>7.23</td>
<td>0.41</td>
<td>98.73</td>
<td>1.06</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>5</td>
<td>14.40</td>
<td>0.81</td>
<td>99.91</td>
<td>0.42</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>10</td>
<td>8.98</td>
<td>0.57</td>
<td>99.61</td>
<td>0.62</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>25</td>
<td>7.85</td>
<td>0.59</td>
<td>99.53</td>
<td>0.87</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>50</td>
<td>7.62</td>
<td>0.40</td>
<td>98.95</td>
<td>0.90</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0.4</td>
<td>15</td>
<td>150</td>
<td>7.62</td>
<td>0.50</td>
<td>98.44</td>
<td>1.00</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>6.99</td>
<td>0.59</td>
<td>98.56</td>
<td>1.36</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>7.01</td>
<td>0.68</td>
<td>97.88</td>
<td>1.34</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>7.48</td>
<td>0.94</td>
<td>98.88</td>
<td>1.24</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>6.85</td>
<td>0.81</td>
<td>98.33</td>
<td>2.39</td>
</tr>
<tr>
<td>17</td>
<td>24</td>
<td>0.4</td>
<td>15</td>
<td>100</td>
<td>7.24</td>
<td>0.90</td>
<td>98.29</td>
<td>2.21</td>
</tr>
</tbody>
</table>

Here experiments show that beyond a lifespan of 10 there is no significant increase in performance.

Experiments 8 to 11, 3 and 12 show that increasing the amount of growth systems decreases the amount of false positives for class 1 (in this case significantly up to \(n_g = 50\) and then smoothly) and slightly lesser the anomaly detection success for class 2. As shown in Equation 8.4, more growth systems can only give better chances for class 1 samples to bind. The drawbacks are again to also offer more opportunities to bind somewhere to some class 2 samples.

Experiments 3 and 13 to 17 show that in this configuration varying the data introduction rate only does not have a significant impact on the organism’s performance. Indeed for class 1, \(p_{r_i} = \frac{25}{26}\), significantly reducing \(p_{na,cell(i)}\) from Equation 8.7 even with a low data rate, making the rate impact less significant.

Comparing these results with the ones from [Bentley et al., 2005] given in Table 8.2 and looking at the best setups, the organism’s results clearly outperform their results. It is interesting to notice that the best setups here use a threshold of 0.4 against 0.2.
in [Bentley et al., 2005]. It seems that for this study, having a low threshold in a deterministic program as in [Bentley et al., 2005] is better whilst in a stochastic approach such as SC a larger threshold works better.

Table 8.2: Percentage of the average of data from each class from the breast cancer dataset creating a danger signal (false positive for class 1 and true positive for class 2) for the two algorithms presented in [Bentley et al., 2005] and using various setups (see [Bentley et al., 2005] for details about the experiments setup).

<table>
<thead>
<tr>
<th>Exp</th>
<th>Algorithm 1</th>
<th></th>
<th>Algorithm 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class 1</td>
<td>Class 2</td>
<td></td>
<td>Class 1</td>
</tr>
<tr>
<td>1</td>
<td>10.5</td>
<td>98.1</td>
<td>20.6</td>
<td>99.99</td>
</tr>
<tr>
<td>2</td>
<td>10.5</td>
<td>95.6</td>
<td>21.2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>10.5</td>
<td>92.3</td>
<td>22.4</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>14.4</td>
<td>99.9</td>
<td>42.3</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>8.9</td>
<td>95.7</td>
<td>15</td>
<td>99.82</td>
</tr>
<tr>
<td>6</td>
<td>22.6</td>
<td>98.3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>8.3</td>
<td>98.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>10.9</td>
<td>99.6</td>
<td>67.1</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>9.9</td>
<td>97.2</td>
<td>14.9</td>
<td>99.98</td>
</tr>
</tbody>
</table>

B. Looking into the organism

Looking into the organism’s inner state over time should provide information about its self-organisation ability (e.g. how big the organism grows, data distribution) and homeostatic behaviour (e.g. danger signals emission, constant size). It is also expected that observing the organism from within should reveal information about the dataset (e.g. danger signals should occur upon malignant data, data should organise based upon similarities between them). To be an effective and useful tissue algorithm, suitable for use with artificial immune system algorithms, the organism should organise itself in a stable pattern reflecting the data distribution while emitting danger signals when appropriate [Matzinger, 1994]. The aim of this experiment is to observe over time the various quantities of systems present within the organism.

Setup

The following experiment uses the setup of experiment 3 from the previous section: a universe, an organism, a time system, a waste recycling, an absorption, a cell recycling, an expelling system, 250 data systems, 100 growth systems. Each data system contains a decay and a split system. The organism initially contains 5 adhesion surfaces. The hunger parameter was set to 200, the threshold to 0.4, the cell’s lifespan to 15. One run is conducted and consisted of 5000 iterations with randomly picked data presented each iteration.
Results

The relevant values to observe along the computation are the amount of cells, clusters of cells and danger signals. Similarly to a real case on-line program execution, Figure 8.4 shows the state over time of the organism during the run. The computations corresponding to the organism’s early life are shown here in order to illustrate the self-organisation of the organism growing to a stable state from a minimal initial state.

Figure 8.4: Organism’s inner organisation over a run of 5000 samples on the UCI breast cancer dataset. The two plots have the same horizontal time axis but differ in the vertical axis scale for readability.
Figure 8.4(a) shows that the organism reaches after about 100,000 computations its mature state. The size and data systems distribution is not specified and is therefore emerging from the organism’s setup (e.g. absorption, recycling and expelling rates) and input dataset. Here the organisms grows to about 25 cells and keeps about 60 adhesion surfaces. The total amount of absorbed systems kept inside the organism in any state (i.e. alive cell, dead cell, adhesion surface, danger signal) is stable around 125 systems. The average amount of cells per adhesion surface oscillates around 10. Once the mature state reached the organism maintains a constant amount of cells in spite of constant cellular death and emission of danger signals. Danger signals are regularly emitted and represent the detected supposedly malignant cells that could then be used by an artificial immune system algorithm. Note that the hunger parameter was set to 200 and the organism settled at around 25 cells, therefore indicating that the size was not impacted or limited by this hunger parameter. Similarly the total amount of available data systems was set to 250 while the amount of used data systems in the model settled around 125, indicating that there was no shortage of data systems.

The organism therefore demonstrates self-organisation by growing from an initial state to a mature state absorbing data as food and converting it into new material while expelling old material. The organism also demonstrates homoeostatic behaviour by maintaining a stable state along computation in spite of malignant data.

Figure 8.4(b) shows the amount of clusters over time (in a smaller scale along the Y-axis for clarity). Results from Table 8.1 already provided insights regarding the inner shape of the organism (i.e. its inner organisation). Class 2 (malignant) items are very well identified which means that class 2 data are not easily aggregated to other data (otherwise they could not be well detected). Therefore class 1 (benign) data only is actually binding together and it is thus expected to observe on average one main cluster all along the program execution. The evolution of the number of clusters indeed shows that the organism keeps settling down into one cluster. New clusters are constantly created with the appearance of new adhesion surfaces but quickly these new clusters bind to the main one.

8.4.2 Analysing Self-Organisation Further

The organism was designed to self-organise data in a data stream based on similarities. In the last set of experiments, the emphasis was on the anomaly detection capabilities of the organism using a dataset with one class of benign data and another class of malignant data. The next two sets of experiments investigate further the data-wise self-organisation within the organism using two datasets containing three classes of data
linearly separable from each other (i.e. no malignant data class and therefore no class should only generate danger signals). First a simple artificial dataset is used to illustrate an ideal dataset’s directed organism’s behaviour. Then the UCI wine [Forina et al., 1991] dataset is used in order to provide a real case study with more noisy data.

A. Ideal Dataset

For this set of experiments a dataset of three classes, compact, linearly separable from each other, and well separated from each other in space was generated. The aim is to have the organism organise itself into three clusters, each holding data from just one class.

Dataset description

Each class contains 50 samples of 3 real-valued vectors. The vectors are randomly generated around the three points \((0.9, 0.2, 0.3), (0.1, 0.8, 0.1)\) and \((0.2, 0.3, 0.9)\) within the squares centred in these points and of side size 0.2, as illustrated in Figure 8.5. Therefore the maximum distance between two points of a class is \(\sqrt{0.2^2 \times 3} = 0.3464\).

![Figure 8.5: Ideal dataset’s data repartition](image)

The minimum distances between any two elements of different classes are between classes 1 and 2 \(0.737\), classes 1 and 3 \(0.6445\) and classes 2 and 3 \(0.6658\).
Setup

A threshold of 0.4 is used to ensure that all data from a same class could bind together and that no data from two different classes could bind together (the minimum distance between two elements of different classes being 0.6445). (Note that the choice of the threshold to this very value is based on the knowledge of this dataset. Indeed, when using an unseen dataset with no such knowledge about its data, it is not possible to know in advance what threshold is best.) Experiments settings use a universe, an organism, a time system, a waste recycling, an absorption (thus data introduction rate is one), a cell recycling, an expelling system, 200 data systems. Each data system contains a decay and a split system. The organism initially contains 5 adhesion surfaces. The cell’s lifespan was set to 15.

4 experiments were run varying the amount of growth systems (25, 50, 75 and 100) in each experiment as Equation 8.4 shows that in this stochastic context the more growth systems, the more likely a cell that should bind to another one will actually do so. The various experiments’ results should therefore experimentally verify this mathematical result as this ideal setup (ideal dataset used with $\tau = 0.4$) asserts that no data from two different classes can bind.

Each experiment consisted of 20 runs, consisting of 3000 iterations with randomly picked data presented each iteration. Each class has the same introduction probability (i.e. $\frac{1}{3}$ each).

Results

Table 8.3 shows the results of various experiments. These results are computed by discarding the early computations (here the first 50 000 computations) during which the organism grows to an adult (stable) state and stopping the experiments when the flow of data ends (thus not permitting organism’s death from starvation). Results give in percentage the average and standard deviation of data from each class creating a danger signal. From the data distribution it is empirically known that only cells from a same class can bind together in this setup (threshold of 0.4), therefore only danger signals (i.e. cell not classified) can provide mistakes.

First, it can be observed that an insufficient amount of growth systems does not allow a good and reliable classification ability, as demonstrated by the first experiment where many danger signals are created (data not classified) but not consistently over the runs (significant standard deviation). As indicated by Equation 8.4, the more growth systems, the better the classification ability. This is demonstrated in experiments 1 to 4 showing that increasing the amount of growth systems decreases the amount of errors (danger signals created). However while the amount of errors is very low in experiment 4, it
Table 8.3: Percentage of the average and standard deviation of data from each class from the ideal dataset creating a danger signal for various amount of growth systems \( (n_g) \).

<table>
<thead>
<tr>
<th>Exp</th>
<th>( n_g )</th>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>stdev</td>
<td>mean</td>
<td>stdev</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>0.95</td>
<td>0.45</td>
<td>0.85</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>0.22</td>
<td>0.21</td>
<td>0.18</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>0.05</td>
<td>0.09</td>
<td>0.06</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0.03</td>
<td>0.08</td>
<td>0.03</td>
</tr>
</tbody>
</table>

is not null. Some data cannot bind which tend to mean they are (almost) single, with respect to their class, in the organism and hence their probability \( p_{ci} \) (as investigated in Subsection 8.3.3) could thus potentially be increased. Increasing the data rate could lower even more the amount of danger signals, provided the organism has a sufficient amount of growth systems. The data rate impact will be further investigated in the next set of experiments using the wine dataset.

Figure 8.6 shows a snapshot of the organism visualised during computation in a run of experiment 2 (taken at a mature state at about 50 000 computations). (Visualisation is explained in detail in Chapter 9.) The spheres are systems and the links between them represent hierarchy (scopes). A system encompassing another one thus has a link from itself to the encompassed system. Figure 8.6(a) shows the whole program structure. To analyse the organisation of the organism itself Figure 8.6(b) shows the structure of the organism and Figure 8.6(c) only shows the cells and the adhesion surfaces, clearly revealing the self-organisation in three distinct organs.

The visualisation and the results presented in Table 8.3 are respectively indicating that three distinct classes were found and that their elements were identified with accuracy.

**B. UCI Wine Dataset**

The previous set of experiments highlighted how the organism works, organising itself with respect to its data flow, but the data stream was an ideal (i.e. easy to organise) stream and datasets taken from real cases tend to have more noisy and sparse data. To investigate further the self-organisation in the organism this new set of experiments uses the UCI Wine dataset [Forina et al., 1991]. This set contains three classes of respectively 59, 71 and 48 instances, linearly separable from the two others and standing for three types of wine grown in Italy. Each sample is a 13 real-valued vector. All input values were normalised to lie within the \([0,1]\) interval.

The aim is to explore how effectively the organism can grow so that the three classes of data are coherently organised in its structure. (It should be recalled that the aim is not to produce a clustering algorithm, but to study self-organisation. Another interest is
(a) Whole Program. The left part represents all the unused resource (waste data) located around the universe (red sphere). The part on the right represents the organism (pink sphere) and its content.

(b) Cells and adhesion surfaces within the organism

(c) Cells and adhesion surfaces only

**Figure 8.6:** Organism’s inner organisation over a run of 3000 samples on the ideal dataset. The spheres represent the systems and the links represent the ownership of a system to a super system. (a) shows a visualisation of the whole program, (b) only shows the organism system (from where all links start), its adhesion surfaces (yellow) and cells (green), and (c) shows the clusters by only displaying the cells and adhesion surfaces. The cells on their own have not yet been linked to any existing cluster.
here to produce a way of organising data such that a separate artificial immune systems
can exploit the coherent tissue structure and provide improved homoeostasis.) The
ideal data stream was compact, with no noise, distributed over three dimensions. Here
the dataset comprises thirteen dimensions and is made of real data measures, therefore
making the organisation more complicated and requiring a more precise analysis.

From the first set of experiments, investigating setups and comparing them in Table 8.1,
the threshold stands as the most crucial parameter, assuming the other parameters are
set to values not too restrictive (e.g. cell’s lifespan too short, data introduction rate
too slow, amount of growth systems too low). The threshold is indeed directly linked
to the precision of cells similarities. Therefore particular attention should be given to
this very parameter. However, as seen in Equation 8.7 and Equation 8.8, the quantity
of data present at a time to compare them is important, therefore several introduction
rates should be investigated to observe the impact on the organism’s behaviour.

Setup

32 experiments were run, involving as parameters:

- $\tau$: the data comparison threshold,
- $d_r$: the data absorption rate (i.e. amount of absorption systems),
- $n_g$: the amount of growth systems.

The principally investigated parameter is the data comparison threshold. Finding an
accurate threshold value depends on the dataset (for a vector of 13 values within $[0,1]$, the
maximum Euclidean distance is $\sqrt{13} \approx 3.6$) and also on the preferences of classification
wanted (e.g. minimise the false positives, or maximise the true positives). The threshold
is initially taken at 0.2 and then varied up to 0.8 in three series of 10 experiments (A, B
and C) where the data introduction rate is different for each.

The data introduction rate (i.e. number of absorption systems) is respectively varied
from 1 (the minimum) to 3 in experiments A to C in order to investigate its impact on
a given threshold.

The number of growth systems is fixed at 50 for all experiments except for two additional
experiments in series C (C3a and C3b) where it is raised to 75 and 100. The aim is to
investigate the impact of having more growth systems in an organism with a higher rate
of absorption (rate of 3), hence with more data to compare at a time, than previous
ones (e.g. experiment set A has a rate of 1).
All experiments were run 20 times, with each run consisting of 3000 iterations with randomly picked data presented each iteration. Each class has the same introduction probability (i.e. $\frac{1}{3}$ each). All experiment settings involve a universe, an organism, a time system, a waste recycling, an absorption, a cell recycling, an expelling system, respectively 200, 400 and 600 data systems for introduction rates of 1, 2 and 3. Each data system contains a decay and a split system. The organism initially contains 5 adhesion surfaces. The cell’s lifespan was set to 15.

**Results**

Table 8.4 shows the results of the various parameter setups that were used, disclosing in percentage the amount of danger signals emitted for each class. However an indication of how well the data are organised within the organism would be useful. The amount of unwanted bonds (referred to as *bridges*) within the organism were therefore also measured. A link between a cell of a class $i$ and a surface is considered as a bridge if the surface binds more cells of another class $j$ together than of class $i$. Bridge evaluation is done when cells die and is given for each class in percentage of the total amount of connections cell-surface. As before, all results are computed by discarding the early computations (here the first 50,000 computations) during which the organism grows to an adult (stable) state and stopping the experiments when the flow of data ends (thus not permitting organism’s death from starvation).

From the results, it can be observed that the amount of bridges is inversely proportional to the amount of danger signals. Increasing the threshold makes it easier for cells to bind together, making less likely the emission of danger signals, but also aggregates data less strictly thus increasing the amount of bridges, and vice versa. Looking at the threshold, varying it little can have a significant impact on the binding results, particularly here between 0.4 and 0.5. However the data introduction rate is also crucial, as a low rate does not provide enough data for the organism to function properly, as seen in Equation 8.7 and Equation 8.8, and as shown by experiments $B3$ and $C3$ compared to experiment $A3$. The amount of danger signals can also be lowered by adding more growth systems into the system, as discussed in Subsection 8.3.3, and shown by experiments $C3$, $C3a$ and $C3b$.

Finally, another effect of a low threshold, or slow data introduction rate, only visible for individual runs, is that the organism might die at birth. The organism then does not manage to bind any cell to others at its early stage (if the datastream provides no similar enough data), leading to the recycling of cells into danger signals and the death of the initial adhesion surfaces. From then on no adhesion surface can be created and the organism is dead. It is what is happening in all the 20 runs in experiments $A1-3$, $B1-2$, $C1-2$ and is responsible for the 100% of danger signals. In these cases the threshold is too
Chapter 8 Self-organisation and Homoeostasis within an Artificial Organism

Table 8.4: Percentage of the average danger signals and bridges (links between cells of a given class and surfaces binding together more cells of another class) created per class from the wine dataset for various data introduction rates ($d_r$), threshold ($\tau$) and amount of growth systems ($n_g$) values.

<table>
<thead>
<tr>
<th>Exp</th>
<th>$d_r$</th>
<th>$\tau$</th>
<th>$n_g$</th>
<th>Class 1 Danger</th>
<th>Bridge</th>
<th>Class 2 Danger</th>
<th>Bridge</th>
<th>Class 3 Danger</th>
<th>Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>1</td>
<td>0.2</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>A2</td>
<td>1</td>
<td>0.3</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>A3</td>
<td>1</td>
<td>0.4</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>A3.3</td>
<td>1</td>
<td>0.43</td>
<td>50</td>
<td>88.23</td>
<td>0.56</td>
<td>92.40</td>
<td>0.49</td>
<td>91.74</td>
<td>0.95</td>
</tr>
<tr>
<td>A3.5</td>
<td>1</td>
<td>0.45</td>
<td>50</td>
<td>36.88</td>
<td>2.37</td>
<td>61.97</td>
<td>2.66</td>
<td>59.08</td>
<td>3.99</td>
</tr>
<tr>
<td>A3.7</td>
<td>1</td>
<td>0.47</td>
<td>50</td>
<td>29.20</td>
<td>3.68</td>
<td>51.74</td>
<td>4.49</td>
<td>48.03</td>
<td>6.69</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>0.5</td>
<td>50</td>
<td>9.44</td>
<td>8.85</td>
<td>30.05</td>
<td>10.01</td>
<td>24.69</td>
<td>14.38</td>
</tr>
<tr>
<td>A5</td>
<td>1</td>
<td>0.6</td>
<td>50</td>
<td>1.39</td>
<td>23.78</td>
<td>9.93</td>
<td>36.65</td>
<td>3.57</td>
<td>36.14</td>
</tr>
<tr>
<td>A6</td>
<td>1</td>
<td>0.7</td>
<td>50</td>
<td>0.43</td>
<td>31.95</td>
<td>1.78</td>
<td>47.09</td>
<td>0.69</td>
<td>40.87</td>
</tr>
<tr>
<td>A7</td>
<td>1</td>
<td>0.8</td>
<td>50</td>
<td>0.05</td>
<td>37.20</td>
<td>0.41</td>
<td>44.60</td>
<td>0.37</td>
<td>39.39</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>0.2</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
<td>0.3</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
<td>0.4</td>
<td>50</td>
<td>48.15</td>
<td>0.52</td>
<td>72.48</td>
<td>0.56</td>
<td>71.68</td>
<td>0.58</td>
</tr>
<tr>
<td>B3.3</td>
<td>2</td>
<td>0.43</td>
<td>50</td>
<td>30.30</td>
<td>1.14</td>
<td>58.66</td>
<td>1.60</td>
<td>54.46</td>
<td>1.99</td>
</tr>
<tr>
<td>B3.5</td>
<td>2</td>
<td>0.45</td>
<td>50</td>
<td>20.05</td>
<td>2.52</td>
<td>48.76</td>
<td>2.73</td>
<td>43.46</td>
<td>3.72</td>
</tr>
<tr>
<td>B3.7</td>
<td>2</td>
<td>0.47</td>
<td>50</td>
<td>14.21</td>
<td>3.53</td>
<td>40.21</td>
<td>4.56</td>
<td>33.68</td>
<td>5.79</td>
</tr>
<tr>
<td>B4</td>
<td>2</td>
<td>0.5</td>
<td>50</td>
<td>9.00</td>
<td>6.23</td>
<td>29.69</td>
<td>7.03</td>
<td>23.28</td>
<td>10.56</td>
</tr>
<tr>
<td>B5</td>
<td>2</td>
<td>0.6</td>
<td>50</td>
<td>1.72</td>
<td>19.79</td>
<td>10.07</td>
<td>26.26</td>
<td>3.60</td>
<td>30.35</td>
</tr>
<tr>
<td>B6</td>
<td>2</td>
<td>0.7</td>
<td>50</td>
<td>0.33</td>
<td>28.15</td>
<td>1.80</td>
<td>41.94</td>
<td>0.66</td>
<td>37.41</td>
</tr>
<tr>
<td>B7</td>
<td>2</td>
<td>0.8</td>
<td>50</td>
<td>0.09</td>
<td>33.44</td>
<td>0.43</td>
<td>42.02</td>
<td>0.37</td>
<td>37.47</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>0.2</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>C2</td>
<td>3</td>
<td>0.3</td>
<td>50</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>C3</td>
<td>3</td>
<td>0.4</td>
<td>50</td>
<td>44.98</td>
<td>0.55</td>
<td>71.16</td>
<td>0.58</td>
<td>69.95</td>
<td>0.62</td>
</tr>
<tr>
<td>C3.3</td>
<td>3</td>
<td>0.43</td>
<td>50</td>
<td>38.60</td>
<td>1.14</td>
<td>64.76</td>
<td>1.24</td>
<td>62.43</td>
<td>1.45</td>
</tr>
<tr>
<td>C3.5</td>
<td>3</td>
<td>0.45</td>
<td>50</td>
<td>24.21</td>
<td>1.92</td>
<td>53.82</td>
<td>2.32</td>
<td>50.31</td>
<td>2.82</td>
</tr>
<tr>
<td>C3.7</td>
<td>3</td>
<td>0.47</td>
<td>50</td>
<td>18.85</td>
<td>2.70</td>
<td>47.16</td>
<td>3.18</td>
<td>41.58</td>
<td>4.65</td>
</tr>
<tr>
<td>C4</td>
<td>3</td>
<td>0.5</td>
<td>50</td>
<td>13.31</td>
<td>4.92</td>
<td>34.64</td>
<td>5.20</td>
<td>30.85</td>
<td>7.66</td>
</tr>
<tr>
<td>C5</td>
<td>3</td>
<td>0.6</td>
<td>50</td>
<td>2.97</td>
<td>15.38</td>
<td>13.29</td>
<td>19.68</td>
<td>5.93</td>
<td>23.59</td>
</tr>
<tr>
<td>C6</td>
<td>3</td>
<td>0.7</td>
<td>50</td>
<td>0.58</td>
<td>23.32</td>
<td>2.91</td>
<td>34.99</td>
<td>1.14</td>
<td>32.01</td>
</tr>
<tr>
<td>C7</td>
<td>3</td>
<td>0.8</td>
<td>50</td>
<td>0.10</td>
<td>28.90</td>
<td>0.49</td>
<td>37.29</td>
<td>0.33</td>
<td>33.51</td>
</tr>
<tr>
<td>C3a</td>
<td>3</td>
<td>0.4</td>
<td>75</td>
<td>42.65</td>
<td>0.42</td>
<td>68.00</td>
<td>0.57</td>
<td>65.68</td>
<td>0.80</td>
</tr>
<tr>
<td>C3b</td>
<td>3</td>
<td>0.4</td>
<td>100</td>
<td>33.30</td>
<td>0.67</td>
<td>57.57</td>
<td>0.79</td>
<td>54.64</td>
<td>0.69</td>
</tr>
</tbody>
</table>

strict, the introduction rate too slow, or a combination of both, to bind things together. $B3$ and $C3$ compared to $A3$ indeed show, as discussed in Subsection 8.3.3, that the rate has an impact. However it also occurs in other experiments. This happened 13 times in experiment $A3.3$, 3 times in experiments $A3.5$, $A3.7$, $B3$, $C3a$, 2 times in experiment $C3.3$, and once in experiment $B3.3$ and $C3b$ (so the results for the experiments where it happens rarely are slightly less good as they would be without these runs). This early death is less and less likely to happen as more data is sent into the organism and as the threshold is increased. However, note that it remains stochastic as it occurred in $C3.3$.
and not in $C3$, even though it was more likely to happen in $C3$ than in $C3.3$.

Figures 8.7, 8.8 and 8.9 show visualisations (visualisation is explained in detail in Chapter 9) during computation (taken at mature states at about 50 000 computations) of the configurations from experiments $C3.3$, $C4$ and $C5$. The spheres are the systems and the links represent the ownership of a system to a super system. This visualisation highlights the organisation of the cells class-wise. Within clusters cells are in light grey and surfaces are grey. Danger signals are outside in dark grey. The three setups illustrate three different behaviours.

Figure 8.7(a) illustrates the organism’s behaviour with a strict threshold ($\tau = 0.43$) where an organisation with few bonds and many danger signals can be observed. Figure 8.7(b) displays a less strict organisation ($\tau = 0.5$). It can be observed in Figure 8.8, breaking down per class Figure 8.7(b), that data from different classes start to bind in some places. Figure 8.7(c) illustrates the impact of a large threshold ($\tau = 0.6$), gathering large amounts of data together with few danger signals but creating many bridges. As shown in Figure 8.9, breaking down per class Figure 8.7(c), data from different classes start binding even more. These three configurations clearly illustrate the impact of the threshold. As the threshold increases the organism has more and more links, creating bigger and more interconnected clusters, but data from different classes also get to bind more and more together.

Overall, looking at the experiments in Table 8.4 and visualisations in Figures 8.7, 8.8 and 8.9, class 1 is clearly easier to classify than classes 2 and 3. The classification results for class 1 are better and the visualisation shows that data aggregate better in class 1. Class 1 set is therefore more compact than the other sets, and would thus consistently have a compactness factor higher than the other classes, as described in Equation 8.8.

### 8.4.3 Analysis

Experiments showed that the artificial organism is self-organised and has a homoeostatic behaviour. These features enable anomaly detection and data classification in the diet and the metabolism of the organism. From a minimal initial state the organism grows to a stable mature state using data as food. The size of the organism and the distribution of its systems, while stable, is not defined by any specific parameter and is thus emerging from the organism’s setup and environment (i.e. food available, cell’s lifespan, speed of food absorption and processing).
Figure 8.7: Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C3.3, C4, and C5. (a) displays an organisation with few bonds and many danger signals due to a strict threshold. (b) displays a less strict organisation where data from different classes bind in some places. (c) illustrates the impact of a large threshold, gathering large amounts of data together with few danger signals but creating many bridges.
Figure 8.8: Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C4 highlighting the data organisation with respect to each class. (a), (b), and (c) show the same display as 8.7(b) but with data from respectively only class 1, 2 or 3.
Figure 8.9: Organism’s inner organisation over a run of 3000 samples on the UCI wine dataset with setup of experiments C5 highlighting the data organisation with respect to each class. (a), (b), and (c) show the same display as 8.7(c) but with data from respectively only class 1, 2 or 3.
Improved Naturalness

This work presented a new kind of program with its own metabolism. As opposed to previous work [Bentley et al., 2005] the program models an organism within its environment and provides the organism with an artificial metabolism based on self-organisation and homoeostasis. The program encompasses fundamental properties of living organisms: eating food, using it to grow tissue, and expelling waste. The metabolism also provides innate immunity by modelling Matzinger’s danger theory. These mechanisms enable the organism to grow from an initial state to an emerging mature state and its metabolism keeps it stable as long as food is provided, failing which the organism would die of starvation. In addition the fixed amount of systems guarantees the energy conservation (in terms of resource) in the ecosystem, making the model more realistic.

Conciseness, Compactness and Readability

This model uses extensively the transformation rule of systemic computation (i.e. systems are transformed but not created from nothing or destroyed). As a result data systems undergo a chain of transformation (see Figure 8.2) that models an ecosystem. Systems change state over time but their systemic existence remains unchanged. Each system has a clearly defined state and role, making the model clear, compact and intuitive. This significantly contrasts with approaches such as the stochastic \( \pi \)-calculus in which new processes are created and old processes potentially ended (also several processes can potentially be created out of one) and where the relationships between processes are unclear (as illustrated in Subsection 3.4.3, page 120).

Also the built-in continuous, distributed, local and stochastic computation allows the modeller to ignore the implementation of these necessary computational features (as would be necessary in conventional programming) provided by definition within SC. As a result the implementation focusses on the model itself thus making it clear and concise.

Properties Exploitation Ability

Provided a continuous flow of data, the organism is designed to live thanks to an artificial metabolism. The organism’s metabolism reflects the self-organisation and homoeostasis ability of the model. Self-organisation is obtained from the distributed computation based on local stochastic interactions that compare data absorbed within the organism and bind them together to create tissues. Unbound cells become danger signals while dead cells that had been bound when alive are recycled into adhesion surfaces. Non recyclable material (e.g. adhesion surface or danger signal for which lifetime is elapsed) is expelled and recycled by the environment. Once recycled data systems can be absorbed again as new food. Continuous computation is therefore exploited to create an ecosystem. Note that this form of model (where resource systems must be consumed,
transformed, and expelled, only to be recycled in the environment) is likely to be commonly needed for growth-type models in SC. Figure 8.10 summarises the properties exploitation within the model.

**Figure 8.10:** Properties exploitation diagram for an artificial organism implemented in SC.
8.4.4 Possible Future Work

As the organism is designed to grow to match the data rate, such a program is therefore able to cope with various (unexpected) parameter changes, self-(re)organising with the data flow, and providing information over time regarding detected potentially abnormal data items. When used in conjunction with an artificial immune system algorithm, the good accuracy of detection (albeit with a high false positive rate), and the automatic organisation of similar data, binding them together, should enable excellent performance overall. While temporal information is currently lost because of the stochastic computation of SC, this could be added as additional features of data items, enabling self-organising according to similar timings in addition to data values. One advantage of SC is the simplicity of modelling new stochastic systems, as illustrated especially in Chapters 5 and 6. Therefore an immune algorithm could be added to this model by simply adding two or three new types of system (e.g. B-cell, T-cell, antibody) that would then automatically interact with the existing tissue systems. The homoeostasis could also be aided by the fault-tolerance and self-repair ability demonstrated in Chapter 6. Having robust software can indeed be an important feature for instance in network security to ensure the program can survive even severe damage provoked for instance by hacking. One potential application could be to test the model against real-time applications like controlling data flows for computer security. Further work could also investigate a dendritic cell algorithm [Greensmith et al., 2005] based innate immunity approach, and also investigate the introduction of adaptive immune response into the organism.

8.5 Summary

This chapter investigated the natural properties of self-organisation and homoeostasis within systemic computation. This was performed by introducing and developing the notion of artificial metabolism using systemic computation in order to create an organism built upon emerging self-organisation and homoeostasis. The organism uses an input data flow to grow and maintain itself in a stable state while detecting anomalies and organising its tissues based on similarities between data. The data is also organised in a suitable way for an artificial immune system which could provide an improved homoeostasis ability. This work is inspired by Matzinger’s danger theory and uses the notion of danger signals. Starting from scratch and working on-line the organism is able to aggregate data according to its similarities and can provide danger signals when cells die in an abnormal way for the current organism. The organism proved to be able to detect anomalous UCI Breast Cancer data with better accuracy than in previous work [Bentley et al., 2005]. The study of the evolution over time of the organism with various
datasets showed that its inner organisation reflects the data distribution of the current flow.

The next chapter addresses a new form of analysis by introducing an on-line visualisation framework for systemic computing.
Chapter 9

Visualisation

9.1 Introduction

Over the previous chapters, a systemic computation platform along with three models of bio-inspired systems (genetic algorithm, artificial neural network, artificial organism) were presented. These models were applied to various complex problems and their exploitation of SC properties was investigated. As discussed in Chapter 3, visualisation tools can also improve the ability to analyse the natural properties and behaviour of models. This final chapter thus presents a different approach to the exploitation of properties within SC: the on-line visualisation of SC models.

Understanding the phases and states that natural and complex systems or processes go through over time is a difficult task. We can observe emerging patterns or behaviours [Holland, 1998] in systems such as cellular automata [Wolfram, 2002], neural networks [Holland, 1998], gene regulatory networks [Kauffman, 1993], swarm intelligence [Clerc, 2006] or genetic algorithms [Holland, 1975, Mitchell, 1996], but the steps and building blocks [Mitchell, 1996] leading to this emergence remain hidden in the final result. One way to help understand the information flow, interactions and emergent behaviour in models is by using computer graphics and visualising it using a non-ambiguous mapping between a computational system and a graphical output.

Systemic computation aims at enabling a clear formalism of natural systems and such formalism can provide the grounding for a non-ambiguous graphical representation. This chapter introduces a novel use of computer graphics for the understanding of natural and complex systems as an on-line 3D visualisation of dynamic systems based upon systemic computation. The underlying formalism of SC enables a visualisation focussing on interactions and structure, thus guaranteeing a comparison at a high level of abstraction between models resembling their original natural system rather than a computer
program. A single model might progress towards various possible distinct states, all with a similar memory usage, yet their meaning can be literally different which only a high level of abstraction analysis could reveal (e.g. two similar neural networks trained to recognise two different objects process the information differently even though using memory in the same way as their structure is the same). Similarly, two models can seem to be very different and so could be their memory usage, yet their behaviour could present some strong similarities which again can only be observed at a high abstraction level analysis (e.g. a genetic algorithm and a swarm algorithm recognising the same object could perform a similar data processing even though their structure and thus memory usage is different). Some natural or bio-inspired systems may resemble each other but behave differently whereas some other systems might look different but behave in a similar way. What are the differences and similarities between an ant colony optimisation system and a genetic algorithm both solving a travelling salesman problem? Evolutionary systems might look different, yet they seem to undergo a similar process, using randomness to find solutions to complex problems. Do certain chemical interactions resemble each other in the way they interact? Do certain neural structures resemble each other in the way their structures change over time?

This work describes and illustrates how a systemic computation model can be visualised and analysed in terms of information flow, interaction, organisation and emergent behaviour in order to achieve such analysis and help answer such questions in the future. The following section provides a background to the area of visualisation. The section after presents the visualisation framework, followed by concrete case studies and other interesting visualisation examples. The final section discusses and concludes on the potential of this visualisation.

The work presented in this chapter was published in [Le Martelot and Bentley, 2009c, 2010].

### 9.2 Additional Background

Many visualisation techniques have been developed over the past twenty years to help understand systems or organise large amounts of data.

The field of bioinformatics for instance is commonly facing complex biological problems with large quantities of data such as genome analysis. Methods such as hierarchical clustering [Johnson, 1966] have been widely used in DNA microarray data analysis to find and identify similar patterns between genes such as regulation or function. These methods usually provide dendrograms revealing the distance between data (e.g. gene
similarities). The most similar genes are paired and a hierarchy is built by iterating this process over genes and clusters of genes. To help users browsing through the data, several visualisation software involving hierarchical clustering have been developed to aid in the analysis of genomic data [Eisen et al., 1998, Karypis et al., 1999]. More interactive visualisation tools have been developed to offer an in depth exploration of the data set [Seo and Shneiderman, 2002]. This approach can provide a data set overview as well as detailed local views. It also enables comparison between various clustering algorithms that might provide a different classification. The dendrogram can be combined at the leaves with a colour mosaic indicating for each gene its level of expression [Seo and Shneiderman, 2002]. The analysis of gene sequences can also be performed over time to identify patterns of interest [Hochheiser et al., 2003]. Pathway tools such as [Karp et al., 2002], KEGG pathways [Kanehisa and Goto, 2000], ExPASy [Gasteiger et al., 2003] can be found and aim at representing knowledge that can then be analysed and potentially visualised. Also, popular bioinformatics tools such as Cytoscape [Shannon et al., 2003] provide a platform for the visualisation of molecular interaction networks and biological pathways. The software supports various data formats enabling the visualisation of networks available from external sources. Visualised data can be positioned, labelled, decorated and commented. The software also allows users to enrich its features by the addition of plugins. While such tools enable a representation and an analysis of the complex structure linking the various entities, along with the knowledge associated to them, it does not provide a simulation of how such models actually behave. It represents knowledge and provides tools and methods to exploit it. The aim of the systemic computing visualisation is to provide an on-line visualisation of models that represents the simulation of these models over time. It thus enables an analysis of the interactions between entities and their evolution through time, hence of the models’ actual dynamic behaviour. Such simulations can indeed inform users about the model and results may then be included within models such as Cytoscape’s. Another aim of the SC visualisation is to remain mainly software controlled rather than user controlled in order to enable a generic representation of models and interactions that can lead to comparisons between models of various natures (hence allowing the comparison of models that can be thought as being significantly different but that can be revealed more similar than expected).

Beyond bioinformatics, valuable visualisation techniques have been explored to deal with complex systems or large data sets. Attractor states in complex systems were visualised in [Gröller, 1994] using Advanced Visual Systems, a general-purpose visualisation system based on a data-flow model. [Löffelmann et al., 1998] later investigated two different methods of visualisation of dynamical systems near critical points such as linear dynamical systems or Lorenz system. These two papers investigated critical states of
mathematical systems, providing insights about their flow dynamics. The modelling of a two user system with shared resource and visualisation of the ongoing process for each user in order to understand the way the system works was performed in [Viste and Skartveit, 2004]. This resource sharing problem, seemingly simple, is a non-linear complex system that may be difficult to control. This work discusses how visualising this system can help users to understand its working principle. Software structures were visualised in [Hendley and Drew, 1995] within a 3D world using repulsive or attractive forces between objects, providing an example of program structure representation in a 3D space using force-based data layout. The World Wide Web (WWW) was visualised in [Wood et al., 1995] also using force-based layout in order to provide the user with information about the structure, the organisation and the content of the space being explored. [Benford et al., 1999] also presented visualisations of the WWW structure, browsing history, searches, presence and activities of multiple users in 3D. These two papers highlight the potential of 3D spaces and force-based algorithms for data structure layout and representation with a large amount of data. The use of motion within visualisation was investigated in [Bartram, 1998]. This work focussed on the exploration of new display dimensions to support the user in information visualisation and explains that motion holds promise as a perceptually rich and efficient display dimension. Tree structures were visualised in [Robertson et al., 1991], highlighting the potential of 3D spaces and interactive animation for the human perceptual system.

These methods all address particular problems and highlight the potential of approaches such as force-based layout engines in space or coloured animation. They provide insights regarding the visualisation approaches to consider for the development of a more generic approach to complex systems visualisation aiming at unifying all the complex systems within a single formalism. In this respect [Bosch et al., 2000] introduced Rivet, a visualisation system for the study of modern computer systems. This framework provides various visualisation tools to visualise data, computer programs memory usage, observe code execution on multiprocessors computer and has a flexible architecture allowing users to define their visualisations. However this approach relies on a conventional view of computation, as opposed to natural computation. Memory and computer resource observation might tell us everything that happens at the hardware level, yet the information explaining the states complex systems like bio-inspired systems go through is of a higher level of abstraction, involving interactions between parts of a system leading to some emerging behaviour.

Higher level implementations of bio-inspired systems along with a graphical representation was presented in [Phillips et al., 2006] using the stochastic π-calculus. As discussed in Chapters 2 and 3, the mathematical nature of π-calculus makes the model implementation non-intuitive, unnecessarily complicated and therefore difficult to approach for a
non-specialist. In this respect a graphical notation for the stochastic $\pi$-calculus was introduced to allow a clearer presentation of the models. Yet this visualisation technique suffers the same issues in expressing the interactions and transformations of entities, resulting in an unclear model.

With its origins grounded into natural computation, SC provides a method to model nature-inspired systems in a way that resembles their true nature. The next section explains how SC can be visualised dynamically to represent and follow the flow of information in SC models.

9.3 Visualising SC Programs

Systemic computation provides a distributed and parallel approach where components interact and can be intricately entwined with the other components. SC models contrast significantly with traditional models as the outcome of an SC model results from an emergent process rather than a deterministic predefined algorithm. Thus, visualising a line of code is here irrelevant as computation is transformation, and transformation is the result of interaction. The SC visualisation must thus reveal the interplay between components and their environment, rather than a low level memory usage observation.

This section introduces the visualisations developed for SC models and illustrates them using a sample model before studying more complex biological networks. This toy model offers a simplified fire chemistry applied to fire spreading. This simple network of chemical reactions enables easy to understand illustration cases the reader can come back to when seeking a visual explanation for what a visualisation tool offers.

The visualiser was built on top of the SC platform presented in Chapter 4. A model can therefore be visualised as it runs (on-line visualisation), as illustrated on Figure 9.1. The user can record all interactions and go back in time or replay a previously recorded execution. A frame rate can be applied to control the speed of the execution if going too fast for the user’s needs. For all details related to the SC environment control, see Appendix A.

9.3.1 Sample Program: A Simple Fire Spreading Model

To help introduce and illustrate the visualisation tools presented here, a toy model of a familiar chemical reaction is used: fire. This model was chosen for it allows to illustrate simply and clearly the notions of scope, interaction and transformation within the visualisation framework.
When modelling with SC it is necessary to perform a systemic analysis in order to identify and interpret appropriate systems and their organisation (see Chapter 3). The first stage is to identify the low-level systems (i.e. determine the level of abstraction to be used). Fire is a combustion, hence some fuel is required. Combustion is a type of oxidation reaction, it therefore requires oxygen. Oxidation is an exothermic reaction which means it releases heat energy. Also for fire to exist the temperature must be high enough to cause combustion. A simple model can therefore take fuel (e.g. wood, petrol) and oxidant (e.g. oxygen) as reactants, combusting in a heat context, and producing heat and exhaust as a result of the combustion. This interaction is shown below using the SC calculus notation (see Chapter 3):

\[ \text{Fuel} \rightarrow \text{Heat} \rightarrow \{ \text{Oxidant} \rightarrow \text{Exhaust Heat} \} \]

Some fuel and oxidant interact in a heat context transforming the fuel and oxidant into exhaust and heat. The heat context is not cited in the right term of the reaction as in SC the context remains unchanged during an interaction.
The distribution of the fuel and oxidants in various geographical places can be done using neighbourhood scopes, sharing some fuel and oxidant, therefore making fire propagation from one area to another possible. Finally the fire has to be initiated in an area by an initial heat system (e.g. match, lightning). The model should thus show the fire spreading across the various areas in contact with one another starting from the fire ignition. The whole fire model is part of a global environment (e.g. atmosphere, ecosystem, universe) commonly represented in SC by a universe system. The structure and principle of the model are summarised in Figure 9.2 using the SC graph notation. The dashed arrows indicating systems transformation are merging to emphasize that the interaction of fuel and oxidant results in heat and exhaust as a product of both reactants.

Figure 9.2: Structure and behaviour of a simple fire spreading model. The whole model is part of a universe encompassing neighbourhoods (here 3), sharing fuel and oxidants. The left neighbourhood has an initial heat. The schemata show the interactions happening and the dashed arrows represent the transformation of systems through interactions.

The complete states sequence is given below using the calculus notation:

Initital:  Neighbourhood₁ ( Heat Fuel Oxidant Neighbourhood₂ )
         Neighbourhood₂ ( Neighbourhood₁ Fuel Oxidant Fuel Oxidant Neighbourhood₃ )
         Neighbourhood₃ ( Neighbourhood₂ Fuel Oxidant Fuel Oxidant )

Step 1:   Neighbourhood₁ ( Heat Heat Exhaust Neighbourhood₂ )
         Neighbourhood₂ ( Neighbourhood₁ Heat Exhaust Fuel Oxidant Neighbourhood₃ )
         Neighbourhood₃ ( Neighbourhood₂ Fuel Oxidant Fuel Oxidant )

Step 2:   Neighbourhood₁ ( Heat Heat Exhaust Neighbourhood₂ )
         Neighbourhood₂ ( Neighbourhood₁ Heat Exhaust Heat Exhaust Neighbourhood₃ )
         Neighbourhood₃ ( Neighbourhood₂ Heat Exhaust Fuel Oxidant )

Step 3:   Neighbourhood₁ ( Heat Heat Exhaust Neighbourhood₂ )
         Neighbourhood₂ ( Neighbourhood₁ Heat Exhaust Heat Exhaust Neighbourhood₃ )
         Neighbourhood₃ ( Neighbourhood₂ Heat Exhaust Heat Exhaust )
9.3.2 Graphic Representation of Models

An SC model is a set of systems where some can act as context of interaction between other systems and some can act as scope for other systems. Scope expresses the notion of hierarchy: a system $S_1$ within another system $S_2$ is in the scope of system $S_2$. The graphical representation must therefore show systems, the hierarchy linking them (i.e. which systems are contained within other systems), the interactions between them and the changes these occur. The power of modern computers allows to use fast per frame computed 3D graphics on top of a virtual machine requiring a lot of computing power.

The visualisation framework provides three representations of a model: a 2D graph, a 3D explorer and a 3D informational structure. A global colour scheme is controlled by the user to give each system type (template of schemata and kernel) a different colour. Figure 9.2 shows the colour per system used in the fire spreading model. Also at any time, the internal binary state of a system (schemata and kernel as held in memory) can be visualised in order for instance to check a counter (e.g. age, amount) or a flag (e.g. initialised/non-initialised) variable. However this system data access is not part of the visualisation process but part of the systemic computer as any significant change in internal representation (system’s data) should be reflected in the visualisation as a change in system’s type (therefore colour).

2D Graph

This first view of the visualiser is a standard 2D graph representing the hierarchy of systems and their types only. The graph is updated over time to display systems in their current state or hierarchical changes, both of which may change due to interactions. Figure 9.3 shows the fire spreading model undergoing interactions and transformations. The fire progression through neighbourhoods can be observed between the two states.

3D Explorer

The second view is a 3D graph working like a systems explorer: the user can explore the graph in depth as if zooming onto a particular area of a map to see more details. Each view is planar and the use of 3D enables going deeper in the structure to see more details (similar to a fractal explorer). Each system is represented as a sphere. Each sphere can contain the subsystems’ spheres, and so on as shown in Figure 9.4. Each deeper level can be explored by zooming into a subsystem. This visualisation method thus provides a local view per scope rather than a global view over the whole model (even though the whole model can be seen from the universe view). Note that the top half of the spheres containing subsystems is hidden to allow the camera to see inside.
This view no longer uses arrows to represent hierarchies as in the 2D graph but represents the subsystems physically within their parent systems (i.e. within the sphere representing their parent system). It allows the visualisation to focus on the interactions happening within each scope. This concept of physical space however leads to a limitation: systems can have several parent systems. In the fire spreading model, in order to spread the fire across consecutive neighbourhoods some fuel and oxidant were shared between neighbourhoods, making these systems belonging to several parent systems. Figure 9.2 used overlapping space to represent this sharing of systems, however when systems belong to many parent systems, themselves having constraints of location due to other systems, and so on, it becomes at some point impossible in 2D or 3D view to solve the physical location constraints imposed by the hierarchy. In an abstract space, adding dimensions could solve the problem but in visualisation the amount of dimensions is limited. Another physical space limitation that no dimensional space can solve is the recursivity in hierarchy (e.g. a system $S_1$ inside a system $S_2$ itself inside $S_1$).

The chosen solution was to consider each scope as a dimension and each view is dedicated to a scope dimension. Therefore each scope has its own physical systems representing the corresponding abstract systems within this scope only. Each system is thus represented
with as many graphical instances (spheres) as it has scopes. The full representation in the 3D explorer of a computational system is thus the combination of all its scope-wise instances.

The graphical layout of the systems is handled by a form of force-based layout algorithm [Fruchterman and Reingold, 1991]. The principle here is, within each scope, to push each system away from each other up to a certain distance (margin) while maintaining the systems physically contained within their scopes (hence constrained in spatial locations). The size of the systems is determined depending on the amount of systems in each scope. Both margin and systems’ size can be adjusted by the user by applying a global scaling factor to all systems. Interactions between systems are represented by temporary forces that attract the interacting systems towards their context of interaction. Note that the SC behaviour does not rely on spatial location but only on scopes. Systems are placed in this way to improve clarity of visualisation.

The most global view is the view from the universe, directly or indirectly containing the whole model. In the fire spreading model, within the universe are the four neighbourhoods, themselves containing the combustible or combusted material as shown in Figure 9.5. In this model some systems are represented several times, once in each scope they belong to. Some neighbourhoods indeed share fuel and oxidant which thus have several graphical instances. In Figure 9.5(b) the right neighbourhood has the initial heat, or match (red system). Then in Figure 9.5(c) one pair of fuel-oxidant (green and

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{3D explorer view of the fire spreading model in its initial state showing (b) the top view (universe) and (c) zooming from the universe into the neighbourhood containing the fire ignition (in red).}
\end{figure}
yellow) has been burnt (red and brown) but a burnt pair (red and brown) also appears in the top neighbourhood. These two pairs are the same but they are represented in their two respective scopes. Figures 9.5(d) and 9.5(e) show progression of the fire until everything has burnt (i.e. no more fuel and oxidant).

**Informational 3D Structure**

The previous visualisation enabled an on-line representation per scope that enables an exploration of the hierarchy. However, the per scope view ruled out the uniqueness of systems representation (systems being represented once per scope). The aim of this new visualisation is therefore to provide a representation where the systemic structure of models is maintained (i.e. one system for all its scopes). This final visualisation is a global representation of the model in its current state through an abstract structure floating in a 3D space.
The layout is also handled by a force-based layout algorithm [Fruchterman and Reingold, 1991] that constantly pushes systems away from each other using Coulomb’s law [Griffiths, 2003]. Subsystems are connected by spring-like links to their super systems, keeping them within distance of the super systems using Hooke’s law [Boresi and Schmidt, 2002]. As a result systems sharing a same scope tend to remain close to each other, as can be seen on Figure 9.6(b). Again the spatial position is used for visualisation alone - it has no affect on the SC behaviour and absolute spatial location does not represent any information (relative location does represent information, however).

An anchor, usually a root system (e.g. universe), is grounded at the centre of the space making the whole model centred and keeps, by its structure, systems within distance of the space centre (without that, systems would drift away indefinitely).

Interactions between systems create a temporary spring-like connection going from each interacting system to the context, as can be seen on Figures 9.6(c), 9.6(d) and 9.6(e). As long as these links remain they attract (also using Hooke’s law [Boresi and Schmidt, 2002]) linked systems towards each other. The lifetime of the links can be changed. Links get more and more transparent as they age until they disappear. Once they disappeared, there is no longer an attraction force between the systems until a new link is created by a new interaction.

The changes of systems over time can be recorded and presented similarly to a tree ring view. The colour at the centre is the current state, and the rings around are the successive past states of the system. Figures 9.6(d) and 9.6(e) display the changes of types in systems, showing the initial states of fuel and oxidant as well as their new state of heat and exhaust. Within each system the larger a coloured ring the longer the system remained in the corresponding state. The successive rings also indicates the order in which changes occurred. If the initial state ring of a system is larger than another system’s then it means that the former remained in its initial state longer and thus its transformation occurred later.

Also, an average abstract shape of the model is provided as an envelope of this model. The envelope shows all the interactions that happened over time (during a given time window or the whole run). Each possible interaction is represented as a 3D bendy pipe going from one interacting system to the other and passing in its centre by the context. The width of a pipe (linking two systems through the context) depends on the amount of times an interaction occurred within the recording window. This envelope therefore gives an overview, an average, of all the interactions and transformations a model underwent. Figures 9.6(f) and 9.6(g) show the envelope of the fire spreading model at two stages in time.
Finally, the time since the last involvement in any interaction can be recorded for each system so that systems not involved in any recent computation can be hidden (time occlusion), revealing only the relevant systems. Systems that recently interacted are opaque and they become increasingly transparent over time if they do not interact. This selection of systems changes over time as systems that have not interacted for a long time might interact again at some other time in the execution. This is illustrated...
in Figure 9.7 showing a fire (of a larger size than previous examples to highlight the effect of time) after total combustion with in 9.7(a) everything displayed and in 9.7(b) progressively hiding systems and their hierarchies if they do not interact any more over time.

![Figure 9.7: Large scale fire (ten neighbourhoods) after total combustion using 3D structure views. (a) shows the structure of the whole model with all systems and hierarchy. (b) shows the same state but with systems involvement in interactions over time displayed with transparency. It can be observed that the fire ignition is almost transparent whereas the last neighbourhoods are opaque.](image)

To navigate in the structure’s space, the camera can be moved and rotated in any direction to enable a user to find the best view.

### 9.4 First Case Study: A Bistable Gene Network

In order to demonstrate the utility of the visualisation framework introduced here this section uses a form of gene regulatory network called a bistable gene network. This form of networks provides an unpredictable behaviour hence offering a good model for analysing visual differences in the visualisations of same networks yet displaying different behaviours. The network used here has also been used in [Phillips et al., 2006] in an approach for visualising stochastic π-calculus models. Reusing the same model enables comparison between the SC model and the stochastic π-calculus model (π-calculus being a well-established paradigm) to verify that the two models behave similarly. The aim
is then to show that visualising such model using SC provides all the material for the analysis of the model’s behaviour over time, whether step by step or overall.

First the model is presented. Then the systemic analysis is performed to create the SC model. The following section analyses the model’s behaviour and compares it with the stochastic π-calculus model’s. Finally the SC visualisation and its analysis are presented.

9.4.1 Model Presentation

A bistable gene network is a form of gene regulatory network (GRN). A GRN is a collection of DNA segments (genes) in a cell interacting with each other indirectly through the proteins they produce. These regulations govern the rates at which genes create proteins. A bistable gene network is a GRN with two distinct possible states. The bistable gene network model used here was presented in [François and Hakim, 2004] and is summarised in Figure 9.8.

![Figure 9.8: Bistable gene network obtained by an evolutionary procedure in silico in [François and Hakim, 2004]. Genes a and b can respectively produce proteins A and B at a given rate. Gene b can be repressed by protein A and then produces proteins B at a much slower rate. Proteins A and B can irreversibly bind to form a complex that cannot inhibit any gene and eventually degrades. Two scenarios are thus possible: either proteins A will initially inhibit gene b and proteins A will be abundant, constantly repressing gene b; or gene b generates a large amount of B, systematically binding to proteins A, bringing proteins A to a low level and preventing a constant repression of gene b by proteins A.

In this case study, the network has two genes a and b that can respectively produce proteins A and B at a given rate. In physics a rate would be an amount of proteins produced per second, in this computer model the rate is the probability of production per systemic interaction. In the network, gene b can be repressed (or inhibited) by protein A and then produces B at a much slower rate than if not inhibited. Also proteins A and B can irreversibly bind to form a complex that cannot inhibit any gene. This complex then eventually degrades. From these rules, two scenarios are possible: either proteins
A initially inhibit gene $b$ and proteins $A$ are abundant, constantly repressing gene $b$; or gene $b$ generates a large amount of proteins $B$, systematically binding to proteins $A$, bringing proteins $A$ to a low level and preventing a constant repression of gene $b$ by proteins $A$.

The rates that were used are the ones used in [Phillips et al., 2006]. They are the same as in [François and Hakim, 2004] except for the degrading rates of unbound proteins $A$ and $B$. Table 9.1 provides all the reaction rates.

<table>
<thead>
<tr>
<th>Reactions</th>
<th>Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \rightarrow a + A$</td>
<td>0.20</td>
</tr>
<tr>
<td>$A \rightarrow \emptyset$</td>
<td>0.002</td>
</tr>
<tr>
<td>$b \rightarrow b + B$</td>
<td>0.37</td>
</tr>
<tr>
<td>$B \rightarrow \emptyset$</td>
<td>0.002</td>
</tr>
<tr>
<td>$A + B \rightarrow A : B$</td>
<td>0.72</td>
</tr>
<tr>
<td>$A : B \rightarrow \emptyset$</td>
<td>0.53</td>
</tr>
<tr>
<td>$b + A \rightarrow b : A$</td>
<td>0.19</td>
</tr>
<tr>
<td>$b : A \rightarrow b + A$</td>
<td>0.42</td>
</tr>
<tr>
<td>$b : A \rightarrow b : A + B$</td>
<td>0.027</td>
</tr>
</tbody>
</table>

### 9.4.2 Systemic Analysis

As mentioned earlier, a systemic analysis is necessary to identify and interpret the appropriate systems and their organisation. This model involves genes and proteins so the level of abstraction should be the one of proteins and genes.

A possible approach is to take a system for each gene $a$ and $b$. For the proteins it is less clear as this model involves amounts of proteins. In such case two ways of modelling can be considered: modelling each protein with a dedicated system, or modelling all proteins with a single system that holds a protein quantity variable. In this model what is to be observed are the two possible states of the network: $A$ is abundant or $B$ is abundant. These states are due to the inhibited or not state of gene $b$, and to the creation of protein complexes making proteins $A$ unable to bind to gene $b$. To record the history of proteins $A$ and $B$ interactions, one protein system with a quantity variable is well suited: its amount of interaction with a gene or other proteins system would reflect the quantity held (no interaction if no protein) and would reveal the network’s behaviour. If there are many interactions between proteins systems $A$ and $B$ then proteins $A$ and $B$ are present in large amounts, and thus gene $b$ is not inhibited. Alternatively, if interaction between gene $b$ and proteins $B$ is weak, then gene $b$ is inhibited, and interaction between gene $a$
and proteins \( A \) should be significantly more important than between gene \( b \) and proteins \( B \). Also having many protein systems would add some complexity to the analysis (many more systems to observe) which is here not necessary.

Genes regulation turns the information on genes into gene products (here proteins). DNA chunks (here genes) can be transcribed into a messenger RNA chunk carrying a message for the protein-synthesising machinery of the cell. This process can be approximated by using a RNA context system that makes interact a gene system and its corresponding proteins system in order to potentially increase the amount of proteins. The RNA chunk involved in the production of proteins \( A \) and \( B \) being different, the contexts of interactions between gene \( a \) and proteins \( A \), and gene \( b \) and proteins \( B \) can therefore respectively be RNA \( A \) and RNA \( B \).

Proteins degrade (proteolysis), therefore proteases (enzymes that conducts proteolysis) have to be part of the model. A proteases system modelling all proteases present in the cell is appropriate (as opposed to many proteases systems) as proteases quantity is not involved in this model. Proteins degrade as a result of chemical interactions within a physical space, therefore an approximation can consider the local energy involved in the process as being context of interaction between proteins and proteases. Local energy is then also involved in the binding of proteins \( A \) and \( B \). Finally corepressor and inducer can respectively be the contexts respectively binding and unbinding proteins \( A \) to gene \( b \).

The interactions are summarised in Table 9.2 and illustrated in Figure 9.9.

**Table 9.2**: Bistable gene network’s interactions. \( q \) is the the quantity of proteins. \( \phi_A \) and \( \phi_B \) are respectively the bound proteins of type \( A \) and \( B \). \( \delta \) represents a quantity variation during an interaction. The notation \( (S_1 ( ) S_2) \) indicates that a system \( S_1 \) is in the scope of another system \( S_2 \) and reciprocally. inhibit is a flag indicating the state of inhibition of gene \( b \).

<table>
<thead>
<tr>
<th>Interactions</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene(_a) \rightarrow RNA(_A)</td>
<td>-{ Proteins(_A[q]) (+\delta q) } \rightarrow Gene(_a) Proteins(_A[q+\delta q])</td>
</tr>
<tr>
<td>Proteins(_A[\phi_A, \phi_B]) \rightarrow Energy(_A)</td>
<td>-{ Proteases (+\delta q) } \rightarrow Proteins(_A[q-\delta q, \phi_A-\delta \phi_B]) Proteases</td>
</tr>
<tr>
<td>Gene(_b) \rightarrow RNA(_B)</td>
<td>-{ Proteins(_B[q]) } \rightarrow Gene(_b) Proteins(_B[q+\delta q])</td>
</tr>
<tr>
<td>Proteins(_B[\phi_B, \phi_A]) \rightarrow Energy(_B)</td>
<td>-{ Proteases } \rightarrow Proteins(_B[q-\delta q, \phi_B-\delta \phi_A]) Proteases</td>
</tr>
<tr>
<td>Proteins(<em>A[\phi_A, \phi_B]) \rightarrow Energy(</em>{AB})</td>
<td>-{ Proteins(_B[\phi_B, \phi_B]) } \rightarrow (Proteins(_A[q-\delta q, \phi_A+\delta q]) (Proteins(_B[q-\delta q, \phi_B+\delta q]) (Proteins(_A[q+\delta q, \phi_A-\delta \phi_B]) Proteins(_B[q+\delta q, \phi_B-\delta \phi_A]) Proteases</td>
</tr>
<tr>
<td>(Proteins(_A[\phi_A]) (Proteins(<em>B[\phi_B])) \rightarrow Energy(</em>{AB})</td>
<td>\rightarrow Proteins(_A) Proteins(_B),</td>
</tr>
<tr>
<td>Proteins(_A[q]) \rightarrow Corepressor</td>
<td>-{ Gene(_b) inhibit } \rightarrow (Proteins(_A[q-1]) (Gene(_b) inhibit))</td>
</tr>
<tr>
<td>(Proteins(_A[q]) (Gene(_b) inhibit) ) \rightarrow Inducer</td>
<td>\rightarrow Proteins(_A[q+1]) Gene(_b) inhibit</td>
</tr>
</tbody>
</table>
Figure 9.9: Interactions and structure of a bistable gene network model. (a), (b), (c) and (d) show the potential structural changes. (e) shows the overall interaction patterns: proteins are created from genes in the context of RNA, proteins can bind to form a complex using local energy, they can also degrade using local energy. Proteins A can inhibit gene b using a corepressor and an inducer can unbind proteins A from gene b.
9.4.3 Model Behaviour

To allow comparison and ensure the model behaves like a bistable switch similarly to the stochastic $\pi$-calculus model by [Phillips et al., 2006], the model was run starting with no protein and recorded the evolution of proteins over time. Thirty runs were performed and showed that the network always falls in one or the other of the two states. Figure 9.10 shows the evolution of protein quantities along two representative runs in each simulation (SC and the Stochastic Pi-Machine SPiM\(^1\)) for the two possible states the network can fall in.

These results are similar to the ones presented in [Phillips et al., 2006] in the sense that the switching behaviour is the same.

However, over the 30 runs, the network fell 4 times (13.33\%) in the abundant $A$ case and 26 times (86.67\%) in the abundant $B$ case whereas the stochastic $\pi$-calculus model

---

\(^1\)http://research.microsoft.com/en-us/projects/spim
fell 11 times (36.67%) in the abundant A case and 19 times (63.33%) in the abundant B case, as shown in Table 9.3. This difference in final state frequency can lie in the differences of representation in the two models, the SC model representing all proteins with a single system whereas the stochastic π-calculus model uses a process for each protein. Also the inner mechanisms of the SC machine and the SPiM, like the stochastic computation simulation, may vary, hence resulting in a slightly different behaviour for equivalent models. There is therefore a computational bias between the two models leading to a slightly different probability to end in each final state in the two models.

Note however that the network was evolved in [François and Hakim, 2004] and reused in [Phillips et al., 2006] to be able to remain in two possible stable states differing in the concentrations of proteins A and B. The probability to fall into each of the two states starting from no protein in the network was not part of the study. It was conducted here to provide a full comparison with the stochastic π-calculus model only. Also as the rates used here are the ones used in [Phillips et al., 2006] which as mentioned above are slightly different from those in [François and Hakim, 2004], such comparison was only possible with the stochastic π-calculus model.

To change the distribution of the final states, to get for instance an equal distribution (any other distribution could be investigated), the rate of release of gene b by proteins A could be decreased, thus making more likely for a repressed gene b to remain repressed.

To investigate this and aim here at an equal distribution of the two final states, 30 more runs were conducted by lowering the rate of the reaction $b : A \rightarrow b + A$ from 0.42 to 0.12. The results, presented in Table 9.3, showed a fair distribution of the final states with 53.33% falling into the A abundant case and 46.67% into the B abundant case.

Table 9.3: Bistable gene network’s final state distribution for the two configurations of the SC model and the stochastic π-calculus model. 30 runs were performed, the final states distribution is given as the number of runs falling into each state, and as a percentage. Configuration SC is the configuration used for the SC model, similar to the one used in [Phillips et al., 2006] for which the results are given in configuration SPiM. In configuration SC (0.12) the rate of $b : A \rightarrow b + A$ is lowered from 0.42 to 0.12 to aim at an equal distribution of the two final states.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SC</th>
<th>SPiM</th>
<th>SC (0.12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State A</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>State B</td>
<td>26</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td>Runs</td>
<td>13.33</td>
<td>86.67</td>
<td>53.33</td>
</tr>
</tbody>
</table>

9.4.4 Visualisations

The abstract structure is designed to display the various interactions and transformations that occurred over time in a model. This case study focusses on the envelope, expected to be particularly relevant as the amount of interactions between systems will appear on
it, thus revealing the state and behaviour of the model. Also the universe, not playing a computational role in the model is hidden from visualisations to only display the relevant systems. Figure 9.11 provides in (a) the colour scheme and then explains the model at various stages of progression, first over early steps and highlighting in the end the two possible behaviours.

Figure 9.11(b) shows the network in its initial state. No interaction happened yet and systems are located around the universe (not shown here but would be located at the centre).

Figure 9.11(c) shows the envelope after a couple of interactions. It reveals that proteins A and B were produced (genes a and b respectively produced proteins A and B in the respective contexts RNA_A and RNA_B). Proteins B appear to have been produced more (more interactions between gene b and proteins B in the context of RNA_B) as the visualisation shows more interaction between gene b (sea blue) and proteins B (light blue) in the context of RNA_B (dark blue) than between gene a (green) and proteins A (light green) in the context of RNA_A (dark green) (the pipe between the gene and the protein systems passing through the RNA system, representing the amount of interaction, is larger in the former than in the latter).

Figure 9.11(d) shows that some proteins A and B bound together. An interaction between both proteins in the context of energy (fuchsia) occurred and a hierarchy link between proteins A and B systems appeared (proteins A and B systems are in the scope of each other). There is also a similar production state of proteins A and B as the visualisation reveals a similar amount of interaction between genes and proteins of both type (pipes equally large).

Figure 9.11(e) shows the first degradation of proteins for proteins B as an interaction between proteins B and proteases (red) appeared.

Figure 9.11(f) shows an equal amount of proteins A and B degraded, with similar amount of interactions between proteases and proteins of both type. A higher amount of protein B has been produced and gene b is still not inhibited. This state already shows signs of advantage proteins B are taking over proteins A as the production rate of proteins B is higher than the production rate of proteins A (see Table 9.1) and proteins A can be produced faster than proteins B only if gene b is repressed.

Figure 9.11(g) shows that gene b got inhibited and then released as interactions occurred between gene b and proteins A in the context of corepressor (yellow) and inducer (orange). Proteins B appear still advantaged over proteins A (higher production of proteins B): the switch is most likely going to carry on that way and proteins B would be abundant in the final stage.
Figure 9.11: Bistable gene network visualisation using the 3D structure with its envelope. (a) Colour scheme. (b) Network in its initial state. (c) Envelope after a few interactions, proteins $A$ and $B$ were produced with protein $B$ produced more. (d) Some proteins $A$ and $B$ bound together. (e) Some proteins $B$ degraded. (f) Equal amount of proteins $A$ and $B$ degraded, higher amount of proteins $B$ produced. (g) Gene $b$ got inhibited and then released. (h) Protein $B$ abundant state. (i) Proteins $A$ abundant state.
Figure 9.11(h) shows the protein $B$ abundant state, as expected, and looking very similar to Figure 9.11(g). Both proteins degraded, with slightly more proteins $B$ that degraded compared to proteins $A$. Gene $b$ got little inhibited (very few interactions between gene $b$ and proteins $A$ in the context of corepressor and inducer as shown by the thin pipes around the corepressor and inducer systems compared to others). Proteins $A$ and $B$ bound a lot (many interactions between them in an Energy$_{AB}$ context) and then degraded but proteins $B$ are more numerous so some proteins $B$ remained whereas proteins $A$ all disappeared.

Figure 9.11(i) shows the alternative final state where proteins $A$ are abundant. This state looks different from Figure 9.11(h) where proteins $B$ are abundant. The main difference lies in the amount of interactions involving the corepressor and the inducer systems. Interactions between gene $b$ and proteins $A$ through respectively the corepressor and the inducer occurred the most (bigger pipes), revealing a constant repression and release of gene $b$. In this respect a hierarchy link between proteins $A$ and gene $b$ can be spotted, showing that at the moment of the snapshot gene $b$ and proteins $A$ are in the scope of each other, which means here that they are bound to each other. This repression of gene $b$ led to a lower production of proteins $B$ (fewer interactions between gene $b$ and proteins $B$ than between gene $a$ and proteins $A$) and a total degradation of proteins $B$ by proteases (amount of interaction between gene $b$ and proteins $B$ as important as between proteins $B$ and proteases).

9.5 Second Case Study: A MAPK Signalling Cascade

The bistable gene network was an example of a non-predictable system with a behaviour nevertheless reasonably simple. The second case study is a significantly bigger and more complicated system: a mitogen-activated protein kinase (MAPK) cascade. This network has also been used in [Phillips et al., 2006] and is reused for the same reasons as mentioned in the previous case study.

First the model is presented. Then the systemic analysis is performed to create the SC model. The following section analyses the model behaviour and compares it with the stochastic $\pi$-calculus model’s. Finally the SC visualisation and its analysis are presented.

9.5.1 Model Presentation

The model presented here is a mitogen-activated protein kinase (MAPK) cascade, as presented in [Huang and Ferrell, 1996] and summarised in Figure 9.12.
A protein kinase is a kinase enzyme that modifies other proteins by chemically adding phosphate groups to them (phosphorylation). Mitogen-activated protein kinases are serine/threonine-specific protein kinases that respond to extracellular stimuli (mitogens) and regulate various cellular activities, such as gene expression, mitosis, differentiation, and cell survival/apoptosis [Pearson et al., 2001]. Here, extracellular stimuli lead to the activation of a MAPK via a signaling cascade composed of MAPK, MAPKK (mitogen-activated protein kinase kinase) and MAPKKK (mitogen-activated protein kinase kinase kinase). A MAPKKK is a kinase enzyme which phosphorylates a MAPKK, which itself phosphorylates a MAPK. Reciprocally, phosphatase enzymes can remove a phosphate group from its substrate (dephosphorylation).

\[ \text{Input (E1)} \]
\[ \text{MAPKKK} \]
\[ \text{MAPKKK}^* \]
\[ \text{E2} \]
\[ \text{MAPKK} \]
\[ \text{MAPKK-P} \]
\[ \text{MAPKK-PP} \]
\[ \text{E2} \]
\[ \text{MAPK P'ase} \]
\[ \text{MAPK-P} \]
\[ \text{MAPK-PP} \]
\[ \text{MAPK P'ase} \]
\[ \text{Output} \]

**Figure 9.12:** MAPK cascade: the activation of both MAPK and MAPKK requires the phosphorylation of two sites, MAPKKK is activated by extracellular stimuli named here E1. MAPK-P, MAPKK-PP and MAPKK-P, MAPKK-PP respectively denote singly and doubly phosphorylated MAPK and MAPKK. MAPKKK* denotes activated MAPKKK. E2 denotes the enzyme deactivating MAPKKK*. P’ase denotes phosphatase.

In this case study, each kinase (respectively phosphatase) first binds to a protein and then can either add to (remove from) it a phosphate group or unbind, letting the protein as it is. Table 9.4 summarises this principle with generic reaction equations.

**Table 9.4:** Principle of phosphorylation and dephosphorylation in the MAPK cascade. Prot, K, P’ase and PO\textsubscript{4} respectively stand for protein, kinase, phosphatase and phosphate.

\[ \text{Prot} + \text{K} + \text{PO}_4 \iff \text{Prot.K} + \text{PO}_4 \rightarrow \text{Prot-PO}_4 + \text{K} \]
\[ \text{Prot-PO}_4 + \text{P’ase} \iff \text{Prot-PO}_4.P’ase \rightarrow \text{Prot + P’ase + PO}_4 \]
All possible reactions in this case study are listed in Table 9.5. To simplify the notation, \( MAPKKK \), \( MAPKK \) and \( MAPK \) are from now on respectively referred to as \( KKK \), \( KK \) and \( K \).

Table 9.5: List of reactions involved in the MAPK cascade. Phosphates are only shown here when bound to a protein (i.e. \( KKK^* \), \( KK-P \), \( KK-PP \), \( K-P \) and \( K-PP \)).

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( KKK + E_1 )</td>
<td>( = KKK.E_1 \rightarrow KKK^* + E_1 )</td>
</tr>
<tr>
<td>( KKK^* + E_2 )</td>
<td>( = KKK.E_2 \rightarrow KKK + E_2 )</td>
</tr>
<tr>
<td>( KK + KKK^* )</td>
<td>( = KK.KKK^* \rightarrow KK-P + KKK^* )</td>
</tr>
<tr>
<td>( K+KK-PP )</td>
<td>( = K.KK-PP \rightarrow K-P + KK-PP )</td>
</tr>
</tbody>
</table>

Each action can be performed at a given rate. As in the previous case study, rates are here transcribed into probabilities. In physics a rate would be an amount of enzymes binding, phosphorylation or dephosphorylation per second, in this computer model the rate is the probability for a reaction to happen per systemic interaction. The rates that were used here are the one from the code example in [Phillips et al., 2006], in which all transitions have a rate of \( 1.0 \).

### 9.5.2 Systemic Analysis

As mentioned earlier, a systemic analysis is necessary to identify and interpret the appropriate systems and their organisation (see Chapter 3). This model involves enzymes (kinase and phosphatase) as well as phosphate groups so the level of abstraction should be the one of enzymes and phosphate groups.

A possible and straightforward approach is to use one system per protein (kinase or phosphatase) or phosphate group. The model should then contain as many phosphate groups as necessary to enable the reactions from Table 9.5 to occur without shortage of phosphate groups (this potential situation is not part of this study). Considering that kinases can bind to one (for \( KKK \)) or two (for \( KK \) and \( K \)) phosphates, then the amount of phosphate systems necessary is given by Equation 9.1 in which \(|X|\) is the cardinal number of the given set \( X \) and \( Phosphates \), \( KKKs \), \( KKs \) and \( Ks \) are respectively the set of systems representing phosphate groups, \( KKK \), \( KK \) and \( K \).

\[
|Phosphates| = |KKKs| + 2 \times (|KKs| + |Ks|) \tag{9.1}
\]
The interactions in phosphorylation and dephosphorylation are happening between a kinase (phosphorylated or not depending on the reaction) and a phosphate group provided that the right activated kinase is present (a kinase is activated once it is phosphorylated enough to be a reactant, as determined by the rules from Table 9.5). Activated kinase systems can thus appropriately be considered as contexts of interaction between a kinase system and a phosphate system. During phosphorylation, a phosphate group is bound to a non-activated kinase, therefore phosphate and kinase systems bound to each other should have a relationship that reflects this connection. One way to achieve that is to set the kinase and the phosphate systems within the scope of each other. Eventually, the kinase might get activated (simply phosphorylated KKK or doubly phosphorylated KK). Reciprocally, dephosphorylation unbinds phosphate systems from kinase systems and deactivates activated kinase systems.

However, looking at Tables 9.4 and 9.5 it can be observed that the activated kinase (or the phosphatase) first approaches a kinase it can phosphorylate (or dephosphorylate) but may eventually not create any reaction. The interactions between kinase or phosphatase systems and phosphate systems must therefore take this into account. Since all reaction rates were chosen to be equal to 1.0, each interaction thus has a probability of 0.5 to create a change in the phosphorylation state of the interacting kinase (e.g. probability of 0.5 that a phosphate is bound to or unbound from a kinase and probability of 0.5 that no change occurs). The SC interactions are summarised in Table 9.6.

<table>
<thead>
<tr>
<th>Interactions</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>KKK</td>
<td>E1</td>
</tr>
<tr>
<td>( KKK[*] ( Phosphate ) )</td>
<td>E2</td>
</tr>
<tr>
<td>KKK[*]</td>
<td>Phosphate</td>
</tr>
<tr>
<td>KKK[PP]</td>
<td>K P'ase</td>
</tr>
</tbody>
</table>

To ensure that the context systems select the appropriate kinase and phosphates (the ones bound to each other in the case of dephosphorylation), in the same manner as physical location would indicate which system is bound to which other system, the notion of scope can be used to encapsulate in an abstract space phosphate and kinase
systems that can interact together. Figure 9.13 illustrates the area scope distribution and Figure 9.14 summarises the whole model organisation.

![Figure 9.13: Distribution in scope areas of kinase and phosphates. Kinase systems are shared between areas where they can be activated/deactivated and areas where they can activate other kinase systems. Here two areas are shown, one with a $KKK$ and the other one with a $KK$ that can be activated in two steps by $KKK$ provided $KKK$ was previously activated by $E1$.](image)

A visualisation of a small MAPK cascade is provided for illustration in Figure 9.15. It shows a MAPK model involving two $KKK$s and five $KK$s and $K$s in its initial state using the 3D explorer and the 3D abstract structure.

### 9.5.3 Model Behaviour

For this study, the model is initialised with a configuration comparable to the one from [Phillips et al., 2006]: 10 $KKK$s, 100 $KK$s, 100 $K$s, 1 Enzyme1, 1 Enzyme2, 1 $KK$-Phosphatase and 1 $K$-Phosphatase systems. The amount of phosphate groups is deducted from Equation 9.1: 410 phosphate systems. The amount of abstract local space areas is given by the amount of kinases (one area per kinase system): 210 area systems. All these systems are located within 1 universe system.

To ensure the model behaves like the stochastic $\pi$-calculus model of a MAPK cascade by [Phillips et al., 2006], the model was run with the same initial conditions: all proteins in a non-phosphorylated initial state (left side of the reactions in Table 9.5). The evolution over time of the amount of each activated kinase (simply phosphorylated $KKK$, double phosphorylated $KK$ and $K$) in the system was recorded. Thirty runs were performed. Figure 9.16 shows the evolution of activated kinase quantities in the SC model and in
Figure 9.14: Structure of a MAPK cascade model. Kinase systems can be activated by phosphorylation in presence of the right activated kinase. Activated kinases can activate non-activated kinase of a different kind (KKK activates KK which activates K) thus creating the cascade. The black dashed arrows represent the transformation of systems over computation. The red longer-dashed lines indicate systems that can potentially be the very same system but are represented twice or more in several places for drawing clarity.

The stochastic π-calculus model (results presented in [Phillips et al., 2006]). For the SC model the results are averaged over the thirty runs. The figure shows that the SC model, like the one from [Phillips et al., 2006], behaves as described in [Huang and Ferrell, 1996]: the signal response is increasingly getting steeper as the cascade is traversed.

Also Figure 9.16(a) shows that in the SC model the amount of activated KKK systems averages around 5. Considering that there are 10 KKK systems and that the probability of KKK to be phosphorylated or dephosphorylated is 0.5 leads to Equation 9.2 (where
Chapter 9 Visualisation

(a) Colour scheme

(b) Explorer view

(c) Structure view

Figure 9.15: Small MAPK cascade model (2 KKKs, 5 KKs, and 5 Ks) visualised with the 3D explorer and the 3D structure.

Figure 9.16: Evolution in the MAPK cascade of the amount of kinases over time (averaged over 30 runs) (a) in the SC model and (b) in the stochastic π-calculus model.

KKK* stands for activated KKK and $t$ is a discrete time).

$$|KKK^*s_{t+1}| = |KKK^*s_t| + (|KKKs_t| \times 0.5) - (|KKK^*s_t| \times 0.5) \quad (9.2)$$

At initialisation there are 10 non-activated KKKs and no activated KKK. Iterating the equation over $t$ then settles both values on 5.
9.5.4 Visualisations

As shown experimentally in Figure 9.16, the MAPK cascade functions as an amplifier where the amount of phosphorylated kinases at a level in the cascade increases faster than the amount of phosphorylated kinases at its previous level.

To understand and visualise how this happens the informational structure of the cascade can be visualised with its envelope to show what systems are interacting with which other systems and responsible for which changes.

Also, considering that kinases keep changing their state (non-phosphorylated, simply phosphorylated, doubly phosphorylated), the tree ring view showing the states over time of the systems is relevant to look at. Figure 9.17 shows an example of the three kinases in their activated state. The displayed states reveal through the changes in coloured rings that while the presented KK in 9.17(c) went here successively from non-activated to simply phosphorylated and doubly phosphorylated, both KKK and K oscillated between phosphorylated states (including activated states) in the past before reaching their current activated state.

To focus on the kinases systems and their interactions only, the universe and the area systems are discarded from this set of visualisations. Figures 9.18, 9.19 and 9.20 provide visualisations snapshots over time of a single run of the MAPK model.

Figure 9.18 shows three early stages, after respectively 10, 25 and 50 interactions, and displays the changes in systems states and the envelope. Only systems involved in at least one interaction are shown, the remaining systems being hidden for clarity.

Figure 9.18(a) (after 10 interactions) shows the interactions that occurred since the very beginning of computation. 3 KKKs got phosphorylated, among which 2 were
Figure 9.18: Visualisation over time of the MAPK model at three early stages using the 3D structure with its envelope.
dephosphorylated and then phosphorylated again (as the envelope reveals by showing that more interactions happened with Enzyme1 than with Enzyme2 involving the 2 KKKs respectively located at the bottom and at the right). The activated KKK at the bottom phosphorylated a KK in turn dephosphorylated by the KK-phosphatase. The activated KKK at the right phosphorylated a KK. In Figure 9.18(b) (after 25 interactions), 4 KKKs are activated and phosphorylating KKS. Four groups of KKS and phosphates centred around an activated KKK at the four corners can be observed. So far 4 KKKs are active and 7 KKS are phosphorylated. Finally Figure 9.18(c) (after 50 interactions) shows the progression of the previous state with more and more KKS being phosphorylated, and the first KS being phosphorylated. The amplification effect is visible with each activated KKK locally phosphorylating in turn several KKS (groups of phosphates and KKS gathering around activated KKKs).

From these first three stages a progression of the global phosphorylation can be observed. The amplification effect is observable with increasingly more KKS being phosphorylated while the the amount of activated KKKs remains stable. It is expected from Equation 9.2 that half of the KKKs (5 KKKs) on average would remain phosphorylated at a time. With only one KK-phosphatase to counterbalance the effect of several activated KKKs, phosphorylation of KKS is inevitably more likely to occur than their dephosphorylation. The same phenomenon is therefore expected to occur on the next cascade level with even more activated kinases (more activated KKS than activated KKKs), leading to a faster phosphorylation of KS.

To investigate this, Figure 9.19 shows two later stages after respectively 100 and 150 interactions, following the stages from Figure 9.18, and illustrating the fast phosphorylation of KS. Note that the changes in systems states (tree ring view) are no longer shown as the global view of the model is getting too large to make it readable at this scale.

Figure 9.19(a) (after 100 interactions) shows the evolution since Figure 9.18(c) with more phosphorylated KKS, several activated KKS (doubly phosphorylated) and KS being phosphorylated in various places. Figure 9.19(b) (after 150 interactions) shows the fast phosphorylation rate of KS being now as numerous as phosphorylated KKS. The notion of local contribution to the global amplification effect well visible in Figure 9.18(c) for activated KKKs is again visible for the next cascade level as groups of KS now also gather around activated KKS. As expected the activation of KKKs remains stable and the phosphorylation of KKS progresses but slower than the phosphorylation of KS. With each activated kinase phosphorylating several kinases of the next level which in turn phosphorylate several kinases of the level after, there seems to be an exponential
Figure 9.19: Visualisation over time of the MAPK model at two later stages (following those from Figure 9.18) using the 3D structure with its envelope.
phosphorylation effect. Further stages are expected to see a significant increase in phosphorylated $K$s and final stages should contain a vast majority of doubly phosphorylated $K$s. Figure 9.20 illustrates this by showing two late stages taken after 250 and 475 interactions.

Figure 9.20(a) (after 250 interactions) shows that more phosphorylated and especially doubly phosphorylated $KK$s and $K$s appeared. At this point there is a similar amount of simply phosphorylated $KK$s (26) and $K$s (27), and doubly phosphorylated $KK$s (21) and $K$s (22). From the results presented in Figure 9.16(a), it is at about 250 interactions on average that phosphorylated $K$s become more numerous over phosphorylated $KK$s. Finally Figure 9.20(b) (after 475 interactions) shows a late stage where all (100) $K$s are doubly phosphorylated while only 43 $KK$s (and 5 $KKK$s) are activated.

The visualisation of interactions over Figures 9.18, 9.19 and 9.20 thus allowed a visual analysis of the local and emerging global behaviour of the cascade over time.

However, to reach the state of total phosphorylation shown in Figure 9.20(b), it is still unclear for instance what the respective roles of some kinases were (how many were involved doing what). Analysing the envelope after the 475 interactions can reveal which systems did not interact, which did and did what, and help to understand the requirements of the model’s behaviour. Previous figures like Figure 9.19(b) clearly showed that all $KKK$s took part in phosphorylation of $KK$s (the envelope shows traces of interactions with $KK$s for all $KKK$s), but it is less clear regarding the involvement of $KK$s. Figure 9.21 shows envelopes after 475 interactions to investigate the implication of $KK$s in the building of the current $K$s phosphorylation state. Phosphate systems are not displayed for clarity.

Figure 9.21(a) shows the non-activated $KK$s at the 475 interactions stage from Figure 9.20(b) and the ones that were involved at least once in a phosphorylation of $K$s are marked with a red area around them (4 of them). Figure 9.21(b) provides a zoom onto the marked one at the top-right of Figure 9.21(a). The violet envelope branch is going from red (colour of $KK$s) to blue (colour of $K$s), revealing an interaction involving a $K$. Figure 9.21(c) shows the involvement of activated $KK$s (43 of them). The ones in the yellow areas (14 of them) did not contribute to phosphorylation of $K$s (their envelope has no branch going towards a blue colour). From these two figures it can be observed that only $4 + (43 - 14) = 33$ $KK$s were actively involved in the current state. Looking at Figure 9.21(d), displaying all systems but phosphates, shows that only 8 systems (in the grey areas on the side), all being $KK$s, were not involved in any interaction. Therefore out of the 100 $KK$s, 8 were not involved, 33 contributed actively to the phosphorylation of $K$s and thus $100 - (8 + 33) = 59$ $KK$s were alternatively phosphorylated and dephosphorylated without getting to phosphorylate any $K$. Computationally, the more
Figure 9.20: Visualisation over time of the MAPK model at an advanced and at a late stage (after those from Figures 9.18 and 9.19) using the 3D structure with its envelope.
Figure 9.21: Involvement of kinases in phosphorylation after 475 interactions. The state is the same as in Figure 9.20(b), using here different views. (a) shows the non-activated KKs and their envelope, revealing that 4 only (highlighted in red areas) were effectively involved in phosphorylation of Ks. (b) provides a zoom onto the top-right KK in red area on (a): the violet branch of the envelope reveals an interaction (phosphorylation) with a blue (K) system. (c) shows the activated KKs and their envelope. Similarly, activated KKs not involved in phosphorylation do not have an envelope branch going towards a blue system, they are here highlighted in a yellow area. Finally (d) shows all kinases at once. The most involved tend to be located towards the centre while the less involved are moved to the sides. Grey areas highlight the systems, all being KKs, that never interacted.
KKs the higher the probability for each KK not to be dephosphorylated. Indeed the phosphatase system being here single and only able to interact with one kinase at a time, it is thus less and less likely to dephosphorylate a given kinase the more numerous they are. Therefore it is noteworthy that the fact that a kinase is not involved in any phosphorylation does not mean it has no impact of the model’s behaviour.

9.6 Other Interesting Situations

So far the visualisation framework was illustrated with models of a fire spreading, a bistable gene network and a mitogen-activated kinase cascade. Other modelling situations were not encountered and are interesting to look at. The following describes two more situations using special examples not related to the previous models.

9.6.1 Systems grouping

One case is the sorting of systems in a scope based on interaction, similarly to a crowd that would gather into clearly defined groups depending on for instance the debated subject.

In the explorer, interactions between systems create a momentary force that brings these interacting systems and the context closer towards their centre of mass. This enables to make appear groups of systems interacting with one another and to place these groups away from each other in the space of a scope. Figure 9.22 shows an example where five types of systems are initially randomly placed within a parent system and then are moved according to their interactions. If considering for instance people attending a research conference, the universe in red can represent the conference hall, the other systems represent people coming from various universities or companies (the colour can be the field of research).

In Figure 9.22(a) people randomly entered the room and in Figure 9.22(b) people are now chatting. The interactions can be people exchanging words in the context of another person momentarily leading a debate (as illustrated by yellow, violet and blue systems), or it can also be an interaction between people and the room (assuming a presentation is going on in a corner of the room) in the context of a speaker. Green systems represent bored people, having no interest in anything going on.

Figure 9.23 shows views of this example using the structure visualisation.

Figures 9.23(a) and 9.23(b) respectively show the structure of the systems (sharing here similarities with social networks [Wellman, 1997]) after many interactions without
Figure 9.22: Example of systems grouping within a given scope (in red) using the 3D explorer. In (a) the systems (e.g. people, colour could represent their field of research) are randomly placed in an initial state. (b) shows the same systems after some interactions: yellow, violet and sea blue systems interact only with systems of their own colours; light blue and brown systems interact with systems of the same colour but also with the parent system (in red); green systems are not interacting at all.

and with time based occlusion. The disappearance of green systems (bored people) in Figure 9.23(a) shows that they did not interact recently, and Figure 9.23(d) shows using the envelope that they did not interact at all. Figure 9.23(d) also shows that yellow, violet and sea blue people respectively interact with all people of their field whereas light blue and brown people also interact with the conference hall, suggesting that people are all debating a lot within their group.

9.6.2 Recursive hierarchies

Another interesting case can be related to the modelling of fractal sets. Considering a fractal set such as the Mandelbrot set [Mandelbrot, 1980], the set is contained within itself. (The set is technically quasi-self-similar and the strict similarity is true around the Misiurewicz points [Lei, 1990, Milnor, 1989].) When zooming in a model using the 3D explorer the user should therefore be able to explore even a recursive hierarchy. As the explorer shows a scope-wise representation of the hierarchy, it actually unrolls the recursive hierarchy to allow an in depth exploration showing clearly the content of each scope. Figure 9.24 shows the 2D graph of an example involving a recursive hierarchy: A contains B which contains D which contains A. We can consider A as a fractal set containing subset B, itself containing subset D containing the set A itself.
Figure 9.23: Example of systems presented in Figure 9.22 captured during execution. (a) shows the structure of the systems without usage occlusion and (b) shows the same state with occlusion. (c) shows the structure of the systems without hierarchy links and without the envelope and (b) shows the same state with the envelope.

Figure 9.24: 2D graph of a model with a recursive hierarchy: A contains B which contains D which contains A.

Figure 9.25 shows the exploration of such model using the 3D explorer. Looking at Figure 9.25(d) it can be observed that D (violet) does indeed contain A and its sub-hierarchy, the exact same A sub-hierarchy that is visualised in 9.25(b). Note that such
Figure 9.25: Example of exploration in depth starting using the 3D structure from (a) the top view (universe), then (b) zooms from the universe into $A$ (green), (c) zooms further into $B$ (blue) and (d) zooms even further into $D$ (violet).

exploration could in theory go on forever (in fact, only up to computer floating point precision), similarly to a fractal set explorer.

9.7 Analysis

The two case studies showed how SC models can be analysed by exploiting the on-line and the overall computational past’s visual information instead of reading and deciphering it from text. They illustrated how behaviour can be tracked and analysed by looking at models undergoing computation.
Improved Naturalness

SC has been designed to imitate natural computation and enable computational models that reflect as much as possible their original concept. Part of this is achieved by using scopes for relative location and influences between systems, and interaction and transformations for the changes in systems’ state. By visualising these concepts the visual output’s interpretation benefits from intuitive cues such as systems’ physical location to express the relations between them, and links between systems to represent hierarchy and interactions. The visualisation of an SC model is thus a concrete output representing an abstraction of the original concept that reveals its underlying computational processes whilst keeping a similar organisation.

The MAPK cascade model illustrated a network where all entities are represented. Visualisations of these interacting entities provided an intuitive representation that shows kinases physically organised and linked depending on their phosphorylation state and located mostly near the other kinases they interacted with. The artificial organism from Chapter 8 is another example of visualisation displaying computation while providing information about the physical layout of the model as computation goes. The bistable gene network focussed on genes activity and thus modelled proteins of each type as one system (there was no need for this study to populate the model with many protein systems). The visualisations thus represented the intensity of the genes’ activity where activity and state changes can be observed using intuitive clues such as proximity and links between interacting systems, and physical bounds for systems hierarchy.

Contrarily, approaches such as \( \pi \)-calculus maintain separate processes where one compound (e.g. a gene bound to a protein) is actually being modelled. Also the use of channels does not reflect a feature of the model but a leftover from concurrent processes communication modelling. This is illustrated in Figure 9.26 providing the graphical representation for the stochastic \( \pi \)-calculus model of the bistable gene network from [Phillips et al., 2006].

Conciseness, Compactness and Readability

As discussed in Subsection 3.4.3 (page 120) and above, SC models provide advantages of clarity compared to other modelling approaches such as \( \pi \)-calculus. The binding of systems, whether genes and proteins in the gene network or phosphate groups and kinases in the MAPK cascade, is modelled in SC within the structure of the models as structural changes occurring over time and resulting from interactions.

Due to its unnatural rules not suitable for biological processes, the stochastic \( \pi \)-calculus model requires two separate diagrams (see Figure 9.26) both representing a part of the
model, leading to an unclear representation. Representing changes of structure in such model is therefore equally unclear.

By using the notions of scopes, interacting systems and transformations SC avoids these issues (e.g. two interacting systems binding to each other are transformed so that they become bound by their scopes) and enables a better consistency between the model and its original concept. Visualising these features thus enables clearer and more intuitive representations of the underlying processes.
Properties Exploitation Ability

Rather than considering models as computer programs, SC considers them as processes where the physical implementation is irrelevant and only interactions and structure matter. By considering these processes from such angle, emphasis is put on their behaviour. The visualisation of these processes hence becomes a graphical mapping of their behaviour and therefore of their displayed properties. Specific systems can be shown or hidden to highlight the activity of a subset of systems. The past states of systems can be visualised as a tree ring view. The overall interaction activity can be summarised by the abstract envelope. Computation can be broken down into steps to visualise progressively the changes occurring in the model, as was performed within the two case studies. Visualising SC is therefore another way in which the natural properties exploited in SC can be analysed.

Local, stochastic and distributed computation is clearly visualised with the ongoing interactions attracting interacting systems towards their context of interaction while the emerging behaviour can be observed within the progressive changes in structure, the respective systems location, the links between them and the envelope. The bistable gene network illustrated a network with several possible states depending upon stochastic interactions. These interactions could be visualised over time and summarised using the envelope, providing a different pattern for each potential final state. This pattern therefore reveals the distribution of interactions. Systems occlusion, the tree ring view and the envelope were used to visualise how interactions happen in the MAPK cascade, displaying how kinases get phosphorylated and involved over time in the phosphorylation of other kinases. Self-organisation has also been visualised similarly within the artificial organism in Chapter 8.

9.8 Conclusion

This chapter presented a set of novel visualisation tools combined with systemic computation for the understanding of natural and complex systems as an on-line visualisation of models using the systemic computation paradigm. The framework involved a 2D graph, a 3D explorer and a 3D informational structure with a graphical on-line trace of the flow which provides a graphical output depending on models’ behaviour. This trace reveals the quantity of interactions that occurred over a window of time or a whole run. The various changes systems underwent are also recorded and presented as a tree ring view.
Two concrete bio-inspired models, a bistable gene network and a MAPK signalling cascade, respectively presented in [Francois and Hakim, 2004] and in [Huang and Ferrell, 1996] and both used in [Phillips et al., 2006], were studied. The bistable gene network, starting from a same initial state, showed after transformations a different shape depending on its final state. The MAPK cascade visualisation provided over time all the interactions information required to understand the inner mechanism for the evolution of activated kinase quantities, increasingly amplified as the cascade is traversed.

The visualisation provides a novel method that reveals the information flow within a dynamic system, allowing analysis of interactions, transformations and structure over time. It provides the control along the information flow to stop and look into a state, varying display options and camera views to analyse the model. The user can go back in time and try an alternative step to see what would happen next if another interaction was to happen. The bistable switch network showed that the final state it ends up in is due to a stochastic combination of interactions that can lead to one or another final state. Only an on-line analysis of these interactions can explain how and when the network switched to a particular behaviour that would lead to its final state. Similarly, only such visualisation of the MAPK cascade enabled the comprehension of the emerging organisation and the distribution of roles within kinases.

The graphical output provides the potential for in depth analysis of more complex models in the future which may have non-intuitive behaviours. Different behaviours could be classified, grouped or identified based on the shape produced by the visualisation. Should two radically different models show the same or similar emergent shapes then both share the same kinds of interactions and information flow. It is anticipated that these kinds of analyses and comparisons could become essential tools for understanding commonalities in biological, natural and complex systems.
Chapter 10

Conclusions

10.1 Summary of Work

Natural processes and conventional computers seem to have one major thing in common: computation. Yet their important characteristics are very different. Natural systems demonstrate many capabilities that are becoming increasingly desirable for our technology. These capabilities are such that biological systems have become highly significant to computer science as examples of highly complex self-organising systems that perform tasks in parallel with no centralised method of control and show homoeostatic behaviour. Based on such observations, the past decades have seen the appearance and development of new major fields like evolutionary computation or artificial life. However, the dichotomy between the properties of natural and conventional computations tends to make these two approaches incompatible. Conventional computers are by essence sequential and rely on a centralised unit of decision, as opposed to natural systems relying on a massively parallel architecture with no centralised control unit. Even though computers are becoming more and more powerful and even parallel, the initial concept of conventional computation is not suited to computation with the natural properties that biological systems have, and hence does not design these machines for a biological way of managing their resources. A biologically inspired approach to the handling and management of modern computer technologies may thus be increasingly desirable. To address these issues and hence provide a way to achieve natural features in computers, systemic computation was introduced as a novel nature-inspired computational paradigm aiming at bringing together natural computation and electronic computation by enabling a digital computer with natural properties. This thesis investigated and developed systemic computation in order to assess its capabilities to provide useful natural features in computer models implemented within systemic computation.
Chapter 1 introduced the context of this work. The notion of computation was discussed, suggesting that natural systems also undergo a computational process. Systemic computation was then introduced as a paradigm and a computer architecture bringing fundamental properties of natural computation within electronic computers. The hypothesis of the thesis was then laid stating that systemic computation enables exploitation and analysis of natural properties within nature-inspired computer models. Definitions of the crucial terms and the thesis’ objectives were then provided. Finally the contributions of this thesis to systemic computation and to the prior work conducted by Bentley were stated, leading then to the thesis’ outline.

Chapter 2 provided a broad literature review locating the work presented in this thesis within the different areas of relevant research. First the theory behind computation and its properties both from a conventional angle, defined in this work as following the von Neumann architecture paradigm, and from a natural angle were discussed. Then hardware and software approaches to natural computation were discussed. Finally alternative paradigms of computation were also reviewed. Each section analysed the contribution of the discussed work to natural properties.

Chapter 3 provided the methodology that has been used in the following chapters for the analysis of natural properties. It defined the properties that would be investigated, the equipment that was developed for this investigation (the version of systemic computation used in this work, the calculus and graphical notations). Then it provided a comparison between modelling nature-inspired systems using SC and using the stochastic π-calculus. This illustrated the significant differences with respect to nature-inspired processes between a paradigm originating in natural computation, and a paradigm originating in mathematics and computer science. It especially showed that SC provides a clearer and more faithful way of modelling such processes. The concept of systemic computing was then presented and compared with conventional computing. The systemic analysis providing the analytical steps used for the modelling in SC was presented. The models chosen for the investigation of the selected natural properties were then discussed. Finally the testing methods used for the analysis and assessment of the work towards providing evidence supporting the hypothesis were presented.

Chapter 4 then introduced the platform that has been developed for this work. This platform is the first complete high level platform implementing systemic computation, including programming language, compiler and virtual machine. All the SC work presented in this thesis has been performed on this platform. All the models were described using the dedicated descriptive SC language, turned by the compiler into bytecode that is read by the virtual machine to execute the user model. The interactions code was
implemented in C++ and compiled as plug-ins loadable by the virtual machine. A command line virtual machine with basic functions as well as a graphical environment with more features were developed.

The following chapters then investigated various natural properties using several implementations of nature-inspired processes.

Chapter 5 presented an investigation of self-adaptation using a genetic algorithm solving the travelling salesman problem implemented on the SC platform. This first study exploited the rules of SC to create a concise GA based on natural rules and evolving solutions one at a time using a one to one comparison. Such approach ruled out any global sorting and provided an implementation significantly different from conventional approaches. The self-adaptation ability of the GA population was tested and experimentally demonstrated against the non-trivial travelling salesman problem. Then flexibility of model evolution was illustrated showing how a feature like self-adaptation of evolution can be added with the minimum of additional code. The new feature was enabled by topping up the GA model with an additional system rather than by redesigning it.

Chapter 6 demonstrated that SC can natively provide programs with fault-tolerant behaviour based on crash-proof computing. To improve this ability the work showed how to easily integrate self-maintenance with minimal software conception overhead. These properties were obtained by exploiting the continuity of systemic computing and the replication of system instances. Fault detection and fault correction are done automatically and are fully integrated into the core of the program. Experiments demonstrated that SC can be a crash-resistant computer able to run fault-tolerant self-maintaining programs in spite of a high probability of fault occurrence and allocating only 10% of its resource to maintenance. The fault detection mechanism is independent from the kind of systems being repaired and can therefore be used in any SC program. Similar to a biological organism, this process is part of the whole and just as any other constituent is a regular and autonomous running task. As such this demonstration also illustrated a form of homoeostatic behaviour.

Chapter 7 showed how exploiting local knowledge and asynchrony within SC can naturally provide more flexibility for artificial neural structures. Contrarily to classical approach, the model gave full autonomy to the neurons in order to enable independence from the structure and therefore not constrain it. Backpropagation was adapted for this model and experiments with this learning algorithm demonstrated that the flexibility is obtained without reducing accuracy.

Chapter 8 investigated self-organisation and homoeostasis with an artificial organism. The artificial organism is an original program involving an artificial innate immune
system. The organism relies on a metabolism that can eat data, self-organise upon these data and expel waste. Using an input data flow as food the organism grows and maintains itself in a stable state while detecting anomalies and organising its tissues based on similarities between data. The data is also organised in a suitable way for an artificial immune system which could provide an improved homoeostasis ability. The organism proved to be able to detect anomalous UCI Breast Cancer data with good accuracy. The study of the evolution over time of the organism with various datasets showed that its inner organisation reflects the data distribution of the current flow.

Chapter 9 introduced a set of novel visualisation tools combined with systemic computation for the understanding of natural and complex systems as an on-line visualisation of models using the systemic computation paradigm. The aim was to provide a graphical output offering potential for in depth analysis of complex models such as natural systems. This was done by developing three forms of visualisation: a 2D graph, a 3D explorer and a 3D informational structure. The latest involved a graphical on-line trace of the flow revealing the quantity of interactions that occurred over a time in a model. Two case studies were investigated: a bistable gene network and a MAPK signalling cascade. They illustrated how the visualisation can reveal the information flow within a dynamic system, allowing analysis of interactions, transformations, and structure over time. The visualisations are hence another way to analyse the natural properties exploited in SC.

10.2 Critical Evaluation

Platform

The platform introduced in Chapter 4 is based upon the thesis version of SC, defined in Section 3.3. While this version was defined for the needs of this work, it discarded features of SC such as fuzzy scopes, partial scope overlapping and partial schemata matching. Therefore this platform does not implement the full set of SC rules defined in [Bentley, 2007a].

Due to the simulation of SC on a conventional computer, real parallelism could not be tested. The simulation of parallelism (using cycles, as defined in Chapter 4, and picking context and interacting systems randomly within a cycle), unavoidable in a simulation context, yet has some drawbacks. Simulating computation that way considers all interactions as if they all require the same amount of time. Also when a system becomes a context system, it is unclear whether it should be added to the list of contexts system of the current cycle, or only of the next cycle, the latter being the choice made in this platform.
The fact that SC is simulated on a conventional computer, not natively suitable for this form of computation, makes systemic computing significantly slower than what a hardware platform is expected to achieve. The SC platform is therefore a valid and valuable proof of concept, but cannot be used for speed performance.

Regarding SC programming the platform uses two languages, one for the model description, one for the code of interaction functions. While this was a fast and efficient solution to provide versatile functions within SC models, it makes the platform harder to use for non computer scientists.

The use of C++ in plug-ins also allowed to load data from input files and collect data into log files. The platform could rely on this possibility but a hardware machine might need to dedicate a part of its architecture to user input-output, failing which the model itself would then need to implement such a mechanism transporting the relevant data from/to an input/output system (e.g. universe). While this has been illustrated in this work (Chapter 6 with a genetic algorithm and Chapter 7 with the artificial neural network), it was not used in all models.

Due to the speed limitations of the virtual machine, the number of experiment runs were limited. Without the speed restriction, comparison could be done using say 100 runs where 10 were performed. Also, due to the speed restriction, the amount of systems handled in a model was limited (to a couple of hundreds, maximum thousands) where a hardware implementation could enable testing with larger amounts (hundreds of thousands, even millions).

Properties

Self-adaptation has been studied using a genetic algorithm solving the travelling salesman problem. Such problem is non-trivial and a significantly hard problem to assess the self-adaptation capability of the GA. Therefore for this study it was deemed appropriate and sufficient. Another approach could have considered a more conventional use of a GA (using genetic operators operating on bits, rather than cities) and adapt mutation and cross-over parameters instead of mutation and cross-over algorithms. The TSP was chosen for it is a well known problem in computational complexity theory and evolutionary computation.

The investigation of fault-tolerance was performed assuming faults were occurring within the schemata and kernels only. The reason for this choice is that the hardware implementation of scopes within a potential systemic computer is yet to be defined. Because of this undefined notion, the choice was made to handle scopes per system as lists that are not accessible from the systems memory (i.e. schemata and kernels). Such lists could abstract a physical implementation of scopes based on hardware reconfiguration.
(e.g. using FPGAs) where a fault in memory does not affect the hardware setup, and a fault in the hardware would make the corresponding system unusable similarly to a system which definition is too damaged for being selected as an interacting system (e.g. characters representing the type of a system damaged). In such case the fault modelling can be considered accurate. However a hardware implementation might choose to implement scopes as a form of schemata matching, and use a similar matching scheme for the identification of systems sharing a scope. In such case the occurrence of faults might yield slightly different situations (i.e. systems changing scopes, or having no scope) that could not be reached in the present implementation. While the GA implementation did not have a crucial need of separate scopes (i.e. solutions and operators are shared in the computation spaces and all are contained within the universe) and hence would be little impacted if at all by scope faults, other models may be affected more significantly. In such case, duplication of scopes and self-repairing scopes could be used, similarly to what has been achieved with the systems’ memory.

Flexibility within artificial neural networks was demonstrated using a backpropagation algorithm. This approach is more grounded into mathematics than in biology, even though this work is concerned with nature inspired processes. This decision was due to a lack of knowledge regarding the learning within the brain, leading to a gap in the model, and hence the need for a working alternative. The choice was made in favour of backpropagation due to its broad use, thus enabling potential for comparison with conventional approaches in order to demonstrate the computational capabilities of the model. Due to a learning method not ideal for the model (i.e. backpropagation was not designed to be accurate with biology), the flexibility demonstration was limited, not by the structure, but by the learning ability. The approach could be exploited significantly better (e.g. learning of massively recurrent networks, development of neural structures) provided a better suited learning approach (i.e. adapting better to any neural structure). However, for the needs of this study, the emphasis was on the neural network model and the induced flexibility, not on the learning paradigm. Also, considering the limitations of the regular backpropagation within conventional use, the adapted version for local and asynchronous computation performed successfully and provided advantages. This version hence also provides a new version of the backpropagation method.

In Chapter 8 anomaly detection was performed using a static data set (UCI breast cancer) and a fixed anomaly introduction rate. For anomaly detection, a non static set with random anomalies and varying introduction rate could have been seemed more appropriate. However the choice of the aforementioned dataset and fixed introduction rate was taken from previous work in order to enable comparison. Also, the data set contained a sparse data distribution for anomalies items, making the set appropriate for the study. In the exploration of self-organisation of data within the organism, some data
classes would overlap, with items of various classes sometimes bound together. A better classification should be possible by trying other comparison functions or adapting the thresholds. However, the purpose of the study was to demonstrate self-organisation, not to optimise the classification ability (e.g. for instance for clustering).

**Visualisation**

While the visualisation tools developed in this thesis have been investigated using two case studies, exploring more models would be useful to ensure enough information is available to the user, and if not what information could be of additional benefit.

Also, the speed of execution of the whole platform makes large models slow to run, hence limiting the scale of runnable models.

### 10.3 Objectives Revisited

The hypothesis of this work is that systemic computation enables exploitation and analysis of natural properties within nature-inspired computer models. To investigate this, eight objectives were listed in Chapter 1. They are now reviewed in the light of the work achieved in this thesis.

**Assess the work done in the related areas and the work done by Bentley with SC.**

Chapter 2 explored the respective foundations and properties of conventional and natural computations. This allowed to understand the underlying concepts of both forms of computations and the reasons why they differ so significantly. The following key concepts of natural computation were defined: *interactions, complexity and complex systems, emergence, architecture of natural systems and self-organisation*. The properties were also defined according to two groups: *computational* for native properties provided by the computational paradigm, and *behavioural* for emerging properties that can result from the exploitation of other properties (whether native or also emerging).

Natural and unconventional computation was discussed in the natural-inspired hardware approaches review presenting alternative models of computers following a non von Neumann paradigm. It was then also reviewed from a more theoretical angle in the review of alternative paradigms of computation.

Biological modelling was reviewed in the nature-inspired software approaches to natural computation review. This section presented the main and most relevant techniques of the field: *evolutionary algorithms, evolutionary techniques, artificial neural networks,*
swarm intelligence, artificial immune systems and cellular automata. Finally the systemic computation work conducted by Bentley prior to this thesis was reviewed. Most of the work had been theoretical and unproven from an empirical point of view. This analysis led to the formalisation of a methodology, discussed in the following objectives, that would be followed in this thesis in order to assess the capabilities of SC.

**Determine which natural properties (computational and behavioural) can be feasibly achieved.**

Chapter 3 analysed properties found in natural systems than can be of benefit for computer technologies provided they are sustainable and obtained without significant overhead. Systemic computation was designed to provide such natural features within a digital computer. Therefore the chosen properties were selected on the one hand for their relevance to natural computation and on the other hand for their interest and potential for digital computing. The properties of self-adaptation, fault-tolerance, crash-proof computing, self-repair, homoeostasis, flexibility and self-organisation were selected for investigation within nature-inspired models. The work presented throughout this thesis provided evidence that these properties are achievable for free or at a low overhead cost using systemic computation.

**Formalise the rules, notation, graphical expression of SC and compare with other approaches.**

Chapter 3 defined the thesis’ version of systemic computation that was used to investigate the aforementioned properties. The rules were very similar to the original rules of Bentley but the thesis’ rules discarded the features not necessary for this study (fuzzy scopes, partial scope overlapping, partial schemata matching). They also specified the undefined notion of recursive scopes by allowing them to be infinitely recursive and contain themselves. Finally the rules allowed the context of interaction to be modified in exceptional cases which benefits for the simulation or the model justify such choice.

SC notations have then been refined. First a calculus notation enables a clear and formal textual notation that describes hierarchies, interactions and transformations. Then a graphical notation is provided to enable intuitive representations of models. Both notations together enable an intuitive and non-ambiguous understanding of the models’ dynamics. The SC notations and SC models structure are compared with the stochastic π-calculus using a nature inspired example modelling an HCl molecule. The stochastic π-calculus was chosen as a comparison for its origins lie in the well established π-calculus, yet provided in this version with stochastic features, necessary for the modelling of natural processes.
The comparison between the natural properties respectively enabled by SC and other approaches (such as various computational paradigms and hardware platforms) has been performed in Chapter 2. The achievement of each property was evaluated for each method, hence enabling comparison at this level between the various approaches.

**Identify the models that can best exploit the selected properties.**

The models that were selected are a genetic algorithm (GA) (in Chapters 5 and 6), an artificial neural network (ANN) (in Chapter 7), an artificial organism (in Chapter 8), a bistable gene network (BGN) and a mitogen-activated protein kinase (MAPK) cascade (in Chapter 9). These models are all nature-inspired but enable the investigation to focus on different natural properties.

Genetic algorithms and artificial neural networks are well established bio-inspired algorithms where SC could contribute additional properties. The GA was used for investigating self-adaptation of the population and then of evolvability using minimal code. It was then reused as a test program to demonstrate fault-tolerance with crash-proof computing and self-repair. This study also provided a case of homoeostatic behaviour.

The ANN was tackled from a biological angle stressing that local knowledge and asynchrony are crucial features that enable flexible neural structures.

The artificial organism is a new kind of program, involving a metabolism, and providing an artificial innate immune system. This model has been used to investigate self-organisation and homoeostasis through its metabolism and growth mechanism.

The bistable gene network and a mitogen-activated protein kinase cascade are two bio-inspired or biological networks. Their aim is not to solve computational problems. They were used as case studies of networks with non-trivial behaviour that could be analysed visually.

**Create a complete SC platform (language, compiler and virtual machine) as a proof-of-concept implementation to carry out experiments on modelling and behaviour of programs within SC.**

As presented in Chapter 4, a complete platform implementing this thesis’ version of SC was developed. It includes a compiler for a dedicated descriptive language used to define the SC models and their allowable interactions. The interactions code is written using C++ and compiled as a dynamic library. The virtual machine loads the model and the associated interaction code plug-in. It then applies systemic computing to the model. The platform also involves a graphical user interface providing more control over the execution of SC programs, and tools for its analysis, among which a visualisation framework.
Perform a systemic analysis and implement those models in SC.

This work contributed to the development of the systemic analysis method, developed in Chapter 3, that aims at identifying and interpreting the appropriate systems and their organisation within a SC model. The systemic analysis is a method for analysing and expressing a given problem or natural system more formally in SC. It was developed to help modellers to think in SC and produce accurate models complying with the logic of SC. In Chapter 3 a simple genetic algorithm was used to illustrate the systemic analysis. The analysis revealed that a GA could easily be modelled with few systems, leading to an implementation which does not involve generations but evolves solutions one by one. The GA was then re-analysed in Chapter 5 for investigating self-adaptation. The analysis revealed that such model can be topped up with additional systems to enable new features. Self-adaptation of evolvability was thus added by injecting one new system into the model. In Chapter 6 the GA was investigated further to study fault-tolerance. The analysis revealed that duplicating systems enables strong tolerance to faults and that again by topping up the model with one new system kind it could become self-repair.

Chapter 7 detailed how a neural network model could be performed within SC. The analysis was performed at the neuron level modelling the notions of dendritic tree, membrane, axon and synapse. This enabled a non-ambiguous signal propagation modelling as opposed to implementations simply modelling neurons and connections between them.

The artificial organism was then modelled in Chapter 8 at the cell level. The analysis lead to an ecosystem containing an organism and its environment. The organism absorbs food systems, turns them into cells, adhesion surfaces (abstraction of sticky adhesion molecules), danger signals (from Matzinger’s danger theory [Matzinger, 1994]) and then expels them as waste which can be recycled within the environment.

The bistable gene network and the MAPK cascade were analysed in Chapter 9. The resulting models were adopting a slightly different approach. The gene network model focussed on the transcription of proteins by genes. To achieve this the choice made was to model all proteins of one type as a single system. The resulting model had few systems and put the emphasis on gene activity. The MAPK model used one system per kinase as the aim of this model is to analyse the cascade process emerging from the phosphorylation of the various kinases.

All these models were implemented on the SC platform described in Chapter 4, and in Chapter 9 for the visualisation part. Each model underwent series of experiments that assessed their level of achievement of natural properties and, when relevant, their efficiency compared to other methods.
Chapter 10 Conclusions

Analyse the exploitation of the properties and assess the advantages and/or disadvantages of using SC for such exploitation and analysis.

Chapters 5, 6, 7 and 8 investigated selected properties within selected models. Each chapter discussed for its respective model in what respects the SC approach provides improved naturalness, conciseness, compactness and readability within the models. A scheme of properties exploitation was also provided, explaining how the investigated properties are achieved within the given model out of native or previously acquired properties. Chapters 9 then discussed how the visualisation of SC allows to visualise and analyse properties within a model on-line.

Investigate visualisation methods for the on-line analysis of SC programs allowing a high-level analysis of information flow and structure within nature-inspired models.

Chapter 9 presented the visualisation framework developed for SC. The focus of these visualisations was put on interactions and transformations within a model in order to provide representations of its dynamics. Such approach enables an analysis of information flow and structure over time at a high level of abstraction, revealing the behaviour of the model. The visualisation framework involves a colour scheme and three representations of a model: a 2D graph, a 3D explorer and a 3D informational structure. The use of colour was chosen in order to differentiate some systems having different definitions (i.e. schemata and kernel). The colour scheme can be set up by the user. The 2D graph is a standard graph representing the hierarchy of systems. It gives a simple and intuitive visualisation of the models’ structure. The 3D graph aims at visualising interactions per scope, considering a scope like a dimension. The aim is to provide a way to explore a model going from scope to scope. By exploring per scope, the explorer method does not provide a global visualisation of systems within all their scopes. This is addressed with the 3D informational structure where the whole systemic structure of models is represented, as well as interactions. Also interactions are recorded in order to provide an average view of past computation, hide systems not involved in computation, and show the successive transformations systems underwent (i.e. changes in colour, thus in definition).

10.4 Contributions

This thesis contributed to the fields of systemic computation, bio-inspired computation, computational modelling and more generally computer science through genetic algorithms, artificial neural networks, artificial immune systems and biological networks.
This thesis provided all the tools necessary to develop systemic computation and investigate its achievement of natural properties:

- Calculus notation,
- Graph notation,
- Comparison of SC with the stochastic $\pi$-calculus,
- Systemic analysis,
- Descriptive programming language,
- Compiler,
- Virtual machine,
- Virtual machine plug-in architecture for user-defined interaction functions,
- Graphical runtime and analysis environment:
  - analyse computation step by step,
  - undo computation,
  - record computation,
  - playback computation,
  - modify universe on-line for parameters passing,
  - see systems value,
  - control computation speed,
- Artificial neural networks runtime environment,
- On-line visualiser.

First, Chapter 2 provided a comparison between conventional and natural computation based on their respective properties. It then provided a review and analysis of the achievement of natural properties by various relevant approaches, whether hardware, software or theoretical.

The behavioural properties of SC given in Table 2.4 (page 108) were only theoretical prior to this thesis as they had not been investigated before. SC was marked full, theoretically, as it aims at providing the properties of natural computation. Table 10.1 provides an update of the achievement of these properties including the contribution of this thesis. Although the final three properties were not explicitly investigated, some of the models actually also display some of these properties. The artificial organism
Table 10.1: Evaluation of the behavioural natural properties achieved by systemic computation. None of these properties had been empirically studied prior to this thesis. The evaluation ranges from no dot, if the property is absent, to three dots if the property is fully supported. An empty dot represents an unproven estimation of how well a property should theoretically or empirically be achieved.

<table>
<thead>
<tr>
<th>Property</th>
<th>Achievement</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-organised</td>
<td>•••</td>
<td>Artificial organism in Chapter 8.</td>
</tr>
<tr>
<td>Homoeostatic</td>
<td>•••</td>
<td>Artificial organism in Chapter 8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Genetic algorithm in Chapter 6.</td>
</tr>
<tr>
<td>Robust</td>
<td>•••</td>
<td>Artificial neural network model in Chapter 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Genetic algorithm in Chapters 5 and 6.</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td>•••</td>
<td>Genetic algorithm in Chapter 6.</td>
</tr>
<tr>
<td>Autonomous</td>
<td>•◦◦</td>
<td>Not explicitly investigated in this thesis.</td>
</tr>
<tr>
<td>Open-ended</td>
<td>•◦◦</td>
<td>Not explicitly investigated in this thesis.</td>
</tr>
<tr>
<td>Complex</td>
<td>•◦◦</td>
<td>Not explicitly investigated in this thesis.</td>
</tr>
</tbody>
</table>

Presented in Chapter 8 is open-ended as its homoeostatic behaviour will never end as long there is food. Also, the models are largely autonomous as while they rely on human input to solve a given problem they do not rely on human input to run successfully (SC is continuous and crash-proof). For instance the artificial organism within its ecosystem is clearly autonomous provided that data (food) is available in the environment. Finally the behaviour of models is relatively complex compared to existing bio-inspired models. Considering again the artificial organism, using only local and stochastic interactions provides an organisation that is not as predictable as an organisation comparing all data one to one as performed in more conventional approaches. Models such as the bistable gene networked illustrated how significant the impact of stochastic computation can be on the final result (state).

The properties of self-organisation, homoeostasis, robustness and fault-tolerance have been investigated. Robustness and fault-tolerance were investigated using different expressions (forms) or levels. Table 10.2 shows for each behavioural property, depending for some of them on their expression, the native properties that were exploited to achieve them. Also, while not explicitly analysed in the work, it should be noted that the models use stochastic, asynchronous, parallel (albeit in simulation), continuous, distributed and local computation natively provided by SC, as well as circular causality built into the idea that two systems affect each other at the same time. (Approximate is not investigated as fuzzy scopes, partial membership and schemata matching was not implemented. Embodiment would require a systemic computer set up within a sensing environment.)
Table 10.2: Properties exploitation scheme showing for each behavioural property what computational property was exploited (marked with dots).

<table>
<thead>
<tr>
<th>Computational</th>
<th>Stochastic</th>
<th>Asynchronous</th>
<th>Parallel</th>
<th>Continuous</th>
<th>Distributed</th>
<th>Approximate</th>
<th>Embodied</th>
<th>Circular causality</th>
<th>Local knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-organised</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Homoeostatic</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robust</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>Self-adaptation</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>Features flexibility</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>Structure flexibility</td>
<td>•</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simply fault-tolerant</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Crash-proof</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Self-repair</td>
<td>•</td>
</tr>
</tbody>
</table>

By developing models suitable for the investigation of these properties this thesis provided novel models of:

- Genetic algorithms, showing how:
  - distributed, stochastic and local computation can lead to a GA without generation but a one to one solutions evolution, and improved self-adaptive features.
  - continuous, distributed and local computation can lead to a fault-tolerant program (here GA) that is crash-proof and self-repair.

- Artificial neural networks, showing how local and asynchronous computation can lead to a model involving a flexible neural structure. This work also provided a distributed asynchronous adaptation of the backpropagation algorithm for such model.

- Artificial immune systems, introducing an artificial organism, a new kind of program with its own metabolism and an innate response based on continuous, distributed and local computation.

- Biological networks providing:
  - a bistable gene network model,
  - a mitogen-activated protein kinase cascade,

that can be analysed in terms of information flow.
10.5 Future Work

Systemic Computation

As was discussed in the criticism section, the management of inputs and outputs in SC had not been defined. The platform in this work used the top system in the hierarchy, the universe, to pass parameters to the program or read returned values from it, but other implementations might make a different choice. Parameters and returned values could be given through an exchange of systems between inside and outside the program. Therefore a study of human-computer interaction with a systemic computer could be performed to help identify how to best handle this communication.

The implementation of the scopes as a list was also an arbitrary choice that could have been made differently. The initial implementation from Bentley was a static fixed size scope table, hence not allowing additional systems to join in. This platform used a different approach by keeping a dynamic list handled per system, hence allowing more flexibility desirable for fully decentralised architecture. However the dynamic list would be of varying size, which depending on the memory allowed per system, might be limited. A study could then also be performed to identify the best solution, or trade-off, with respect to some given hardware.

The SC calculus presented in Chapter 3 could be developed further. One possibility could be a translation to the stochastic $\pi$-calculus to benefit from the algebra research on this method. The SC calculus could also be developed to have its own theorems and properties. Finally the calculus notation could also be developed to become a programming language for SC.

Platform

The platform in this work used a descriptive language to program the SC architecture, and C++ to program the interaction code. Having a dedicated language and compiler also for the code defining interaction functions could enable to reduce the code in the plug-ins by exploiting for instance specific operators and conversion operations performed by the compiler (similarly for example to the compression operator from the SC descriptive language). Having one global language and one compiler for both would also simplify the use of the platform and help control better allowable and forbidden operations (e.g. writing in the context in Bentley’s SC rules).

The platform has been developed as a research tool for building a systemic computer and investigating SC. The code and architecture were then designed first as an evolutive tool with reusable and flexible code, but not fully speed optimised. The SC research presented in this work having reached a stable definition of its (according to the thesis’
rules) virtual machine and language, the platform could then benefit from speed-ups by reprogramming the core. Such reprogramming could also involve the exploitation of available resource in the graphics processor (GPGPU) by using libraries such as OpenCL. In that respect recent work investigated the use of GPUs for SC\(^1\).

The platform could also be extended to implement the full set of rules of SC including fuzzy scopes, partial scope overlapping and partial schemata matching.

The platform is a proof-of-concept implementation with which experiments on modelling and programs’ behaviour within the SC paradigm could be carried. Further development of SC should lead to the creation of a hardware systemic computer. A new project aiming to create a hardware-based systemic computer using FPGAs has just begun\(^1\).

**Properties**

The study of self-adaptation provided a new kind of GA that has the potential to self-adapt at the population level and the evolutionary level. It has been investigated using a genetic algorithm solving the travelling salesman problem. Other problems could also be tested to identify what form of problems or fitness landscapes can particularly benefit from a one-to-one solution evolution GA or from the evolution of evolvability feature.

The study of flexibility has been limited to using a non-biological learning paradigm. As discussed in Chapter 7 the model was designed to accept the addition or removal of connections. Flexibility could thus be taken further by enabling bio-inspired learning within the model that could be better suited for massively recurrent networks, evolving networks, or networks developing their own neural structures involving for instance an artificial metabolism managing the resource.

Demonstrating that SC is crash-proof, fault-tolerant and self-repair has various benefits for reliable computing systems. Programs that must not crash such as autopilot software, or programs that must cope with unexpected situations (robust software) can greatly benefit from these abilities by never halting. Network security programs undergoing hacking can benefit from these properties to resist and recover from damage. Also, when a program is not behaving as expected, or too damaged for self-maintenance, damaged systems can be replaced on-line (e.g. memory reset) without risking terminal failure.

The artificial organism exploited for self-organisation and homoeostasis has been setup using an Euclidean distance and fixed parameters per experiment (e.g. threshold \(\tau\)). The classification and anomaly detection abilities of the approach could be investigated further with different data comparison techniques or parameters. The threshold could

---

\(^1\)More about SC projects can be found at [http://syscom.wikidot.com](http://syscom.wikidot.com).
also be made adaptive based on the data input stream in order to get a strict classification when data classes are compact and a more flexible classification when data distribution is sparse. Also, for anomaly detection, the chosen anomaly introduction rate was chosen from previous work to enable comparison. Other rates, including irregular rates, could be investigated to study the impact of anomaly introduction rate on the detection ability. Finally the artificial organism was also designed as an interface for artificial immune system algorithms. As discussed in Chapter 8, future work could then improve the homoeostasis behaviour by introducing an AIS by adding new types of system. This could be ultimately tested against real-time computer security applications.

**Visualisations**

Because the visualisation framework was built on top of the virtual machine, it suffers from the same speed limits. In addition the graphics also require computing time, slowing down the process. Just as the rest of the platform the visualiser was developed as an experimental and evolvable piece of software, hence not using fully optimised graphics algorithms. To speed up the visualisation, in addition to reprogramming the core of the virtual machine, the visualiser could make better use of dedicated graphics hardware and exploit associated techniques (e.g. shaders) in order to let the graphics processor handle most of the geometrical calculations (e.g. curve equations and envelope).

Several visualisation tools have been developed and their use was investigated with known concrete case studies. Further investigations, for instance within bioinformatics, could tackle models with partly or badly understood behaviour to help understand the dynamics behind the observed phenomena. Other studies could investigate common visual patterns between models and hence reveal visual similarities, suggesting common underlying principles. These additional studies would also help identify, if any, what visual information could be added to the visualisation framework.

### 10.6 Closing Words

This thesis evaluated, investigated and developed the novel nature-inspired computing paradigm systemic computation by focussing on natural properties. The study was concerned with the exploitation of these properties within nature-inspired computer models in order to provide more advanced natural properties, and with their analysis in order to assess the valuable aspects of the approach. A complete methodology has been provided to conduct and assess the work. The first high level SC platform developed for this study was presented. Several models inspired from nature were presented. Each of them was used to study specific behavioural natural properties, exploiting computational
or pre-acquired behavioural natural properties. A visualisation framework providing novel ways of analysing these properties was also introduced. Series of experiments were systematically run to assess the presented implementations. Each study was summarised by an analysis section addressing the various contributions to the global hypothesis, thus providing evidence that systemic computation enables exploitation and analysis of natural properties within nature-inspired computer models.
Appendix A

Platform Environment

A.1 Introduction

The SC platform presented in this work contains a compiler, a command line virtual machine, a graphical runtime environment that can also offer an on-line visualiser for SC programs. It also contains an artificial neural network graphical application. The following sections provide the instructions for using the various platform features. The distribution can be downloaded from http://www.cs.ucl.ac.uk/staff/E.LeMartelot. The project is also hosted at http://code.google.com/p/sc-scope. All binaries can be found in the root directory of the SC platform distribution which has the following content:

<table>
<thead>
<tr>
<th>Directory or File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>Directory where to place SC bytecode files.</td>
</tr>
<tr>
<td>maps</td>
<td>Directory where to place the compression code maps.</td>
</tr>
<tr>
<td>plugins</td>
<td>Directory where to place the compiled C++ plugin modules.</td>
</tr>
<tr>
<td>src</td>
<td>Directory where to place the SC source code files.</td>
</tr>
<tr>
<td>visual</td>
<td>Directory where to place mask files for the visualiser.</td>
</tr>
<tr>
<td>scc</td>
<td>SC compiler.</td>
</tr>
<tr>
<td>sc</td>
<td>Command line virtual machine.</td>
</tr>
<tr>
<td>SCLab</td>
<td>SCLab environment.</td>
</tr>
<tr>
<td>SCLabLite</td>
<td>SCLab environment lite version shortcut.</td>
</tr>
<tr>
<td>SCANN</td>
<td>SC artificial neural network environment.</td>
</tr>
<tr>
<td>GNU/GPL_v3.txt</td>
<td>Text of the GNU general public license version 3.</td>
</tr>
<tr>
<td>Readme.txt</td>
<td>Readme information text.</td>
</tr>
</tbody>
</table>
A.2 SC Compiler

The SC Compiler `scc` takes a list of source files as arguments and can take several options. Usage is as follows:

```
scc [options] src_1 [ src_2 ... src_n ]
```

with arguments `src_1` to `src_n` being `.sc` source files. At least one source file must be provided. The suffix `.sc` does not need to be given. A list of files with a same part in the name can be given using the `*` character (e.g. `prog*.sc`).

The possible options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-t</code></td>
<td>directory</td>
<td>Output directory where to place target files.</td>
</tr>
<tr>
<td><code>-m</code></td>
<td>directory</td>
<td>Character code map file directory.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td></td>
<td>Verbose mode which outputs the compression code, the external plugin names,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the word length, the function offset, the function codes, the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>declarations, instances and scopes.</td>
</tr>
<tr>
<td><code>-h</code></td>
<td></td>
<td>Displays help information.</td>
</tr>
</tbody>
</table>

Therefore, to compile the source file `prog.sc` located in the current directory, using the code map `sc_code.map` located in the `maps` directory and place the generated files into the `bin` directory, the command is:

```
scc -m maps -t bin prog.sc
```

This command would output two files to the `bin` directory: `prog.bsc` and `prog_macros.h`.

- `prog.bsc` is the bytecode file the virtual machine will load. It contains all the machine code for the virtual machine, except the interaction functions which are implemented in separate plugins.

- `prog_macros.h` is a C header file containing the offsets and lengths of the defined labels and function labels as C preprocessor macros. For example, the two lines from a SC source program

<table>
<thead>
<tr>
<th>label</th>
<th>SYSTEM_TYPE</th>
<th>???? 1001 &lt;?</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>Interact</td>
<td>0001 &lt;?</td>
</tr>
</tbody>
</table>
would respectively produce the corresponding pairs of macros

<table>
<thead>
<tr>
<th>#define</th>
<th>SYSTEM_TYPE_OFF 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define</td>
<td>SYSTEM_TYPE_LGH 4</td>
</tr>
<tr>
<td>#define</td>
<td>Interact_OFF 0</td>
</tr>
<tr>
<td>#define</td>
<td>Interact_LGH 4</td>
</tr>
</tbody>
</table>

indicating the offset (suffix OFF) of the definition and its length (suffix LGH).

The purpose of this file is to be included as a source file in the associated C++ plugin module. Using the macros users can access the parts of interest in the systems without having to know the value of their offset or their length. In addition, if the SC code is modified such that the location or size of some labels is changed, the C++ code remains the same and just need the latest output version of the header to be recompiled.

### A.3 Command Line Virtual Machine

The command line virtual machine sc takes a bytecode file as required argument and can take several options. Usage is as follows:

```
sc [options] input
```

with argument input being a .bsc bytecode file. At least one source file must be provided. The suffix .bsc does not need to be given.

The possible options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p or --plug_dir</td>
<td>directories</td>
<td>Plugins directories separated by ‘;’.</td>
</tr>
<tr>
<td>-r or --record</td>
<td>file</td>
<td>Records execution to the given file.</td>
</tr>
<tr>
<td>-h or --help</td>
<td></td>
<td>Displays help information.</td>
</tr>
</tbody>
</table>

Therefore, to execute the bytecode file prog.bsc located in the current directory, using the plugin directory plugins and record execution into prog.log, the command is:

```
sc -p plugins -r prog.log prog.sc
```

The log file provides the result of all iterations that occurred within the virtual machine’s engine. Each line provides the data of an iteration and has the following syntax if a computation occurred:
1. Iteration number

2. Computation number

3. Scope index

4. System 1 index

5. Context index

6. System 2 index

7. Changes made to system 1

8. Changes made to context

9. Changes made to system 2

Changes to systems have the following syntax:

1. Number of changes in schema 1

2. For each change in schema 1
   - Index of change
   - Sub string to put instead

3. Number of changes in the kernel

4. For each change in the kernel
   - Index of change
   - Sub string to put instead

5. Number of changes in schema 2

6. For each change in schema 2
   - Index of change
   - Sub string to put instead

7. Number of changes of super systems

8. For each super system change
   - True if gaining a system, false otherwise
   - System index
9. Number of changes of sub systems

10. For each sub system change
    
    • True if gaining a system, false otherwise
    • System index

If no computation occurred, the log line is then simply:

1. Iteration number
2. -1
3. Context index

The index of a system is its number in the systems array as held in the virtual machine’s memory. All system indices can be provided at compilation when using the mode verbose. They can also be obtained by using the graphical runtime environment SCLab.

A.4 Graphical Runtime Environment

The graphical runtime environment SCLab provides a virtual machine with a GUI that enables control over the execution of a program. SCLab has two modes: lite or full. The lite mode provides the virtual machine data and execution control tools. The full mode consists of the lite mode topped up with a visualisation framework (discussed in Chapter 9). When running SCLab, it is possible to switch from one mode to the other at any time.

The SCLab executable file starts in full mode by default but can be called as follows

```
SCLab [options]
```

with the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l or –lite</td>
<td>Starts in lite mode.</td>
</tr>
<tr>
<td>-h or –help</td>
<td>Displays help information.</td>
</tr>
</tbody>
</table>
A.4.1 Virtual Machine GUI

Figure A.1 shows a snapshot of the lite mode and gives the panel distribution across the GUI. All panels located on the left side and the tool bar can however be dragged around and thus change position in the GUI. The panels can also be resized or dragged out as separate child windows.

Figure A.1: SCLab lite mode
The various panels are detailed below:

- The menu bar and the tool bar contain an access to all the options available in the application panels. In addition the View menu in the menu bar allows to switch to full or lite mode. Using the Config menu or the associated tool buttons the user can set:
  
  - The list of directories where plugins can be found, as with the command line plugin directory option.
  
  - The recording level of the last iteration (2 by default): 0 no system data is recorded, 1 the index of each system in the interaction triplet is recorded, 2 in addition to the system indices all the changes they underwent is recorded. In order to record all data to a log file, or to visualise the interactions on the visualiser, the recording level should be set to 2.
  
  - The buffer size for memorised states. The buffer is a stack of machine states where each state contains the memory and variables of the machine (and potentially visualiser) as they were at the given time. When undoing an iteration, the platform pops the top state of the buffer (current state) and reads the new top one to set up the machine in this state. Note that the buffer has nothing to do with the recording of the last computation.
  
  - The timer of the engine can be set to attempt a given amount of iterations per second. By default, the computer goes as fast as it can. However if computation is going too fast for a user that for instance wants to follow the changes in states on the systems panel or on the visualiser, then the timer can slow down the engine’s iteration rate.
  
  - The About menu provides information regarding the platform version, authorship and copyrights.

- The Program panel from which a program can be loaded indicates the name of the program and its number of systems.

- The Program State panel gives the current state of a program over execution. First the number of context systems which can potentially change over time. Then the number of elapsed iterations (VM engine loop, see Chapter 4) and computations (iterations where a computation did occur). Below the last interaction is given as the first interacting system followed by the context of interaction and the second interacting system. An undo button allows to step back in time of one step at a time. Finally the reset button resets the program to its initial state.

- The program Execution panel enables to let the program run by clicking the run radio button, to run one iteration forward or to run until the next computation.
A record file can also be given in order to log the computation, as discussed in the command line virtual machine section above.

- The program *Playback* panel enables the user to select a log file that was previously recorded from a model, and to play it on the same model. The playback can be let running by clicking the *run* radio button or can run step by step.

- The program *Universe* panel provides the user on-line with the value of the universe and allows to modify its data. This interaction with the program follows the logic of the user-program interaction within the context of the systemic computer, illustrated in Figure 3.2 (page 113). This is also an area where the user can conveniently read output data from the program.

- The program *Systems* panel lists all the systems of the program, providing their index in the systems array (index used in the log to identify systems), their name as defined in the program section of the SC source program, and the value of their schemata and kernel.

- The *Console* panel lists the actions being performed, like loading a program, initialising it, running it or stepping backward (undo).

- The *status bar* indicated the number of systems, the running state (i.e. running, stopped), the number of contexts and the number of computations.

The layout of the full mode is slightly different and is given in Figure A.2. Its tool bar enables the visualiser buttons.

### A.4.2 Visualiser

The visualiser panel provides all the tools developed to visualise on-line computation of a SC model (see Chapter 9). The panel is shown in Figure A.3 and consists of five panels. Note that none of the setup in the visualiser affects the computation itself, it only affects its visualisation.

**Control panel**

The control panel provides control over the visualiser’s engine that records computations and keeps track of them.

- The *program state* box concerns the update of the visualiser with respect to the computation. The *track changes* checkbox enables or disables the computation recording. When this box is disabled the visualiser does not receive the update
events to reflect the computations. In that case the load state button is enabled and can be clicked to load the current state of the model, but hence with no knowledge of prior computations involved in its representation. When the changes are tracked the visualiser is updated at each computation and can take into account the past states into the current state’s representation.
The enable box allows the explorer and structure panels through checkboxes to be enabled or disabled from the visualiser. Visualising computation being a computationally expensive task, the process does slow down the platform and thus the computation rate. It can therefore be useful to disable the visualisations that are not used.

The systems usage sliders set up the usage (i.e. the involvement) of systems during computation. The decay checkbox sets up how fast a system that has been involved...
in a computation is less and less considered as recently used. The raising checkbox sets up how significant the increase of a system’s importance should be when involved in a computation. The less a system is involved, the more transparent it gets until disappearing. The initial value is set to full usage for all systems so that they all equally appear. The clear button resets the usage to default value.

- The changes box allows to store or not past types (as defined by masks) of the systems in order to allow the tree ring visualisation (in the structure) of system’s types over time. If enabled the recording window for types recording can be set up. The window is given in amount of computations and thus keeps track of any type the system had during the past given amount of computations. If set the 0, the window’s length is infinite (within limit of available memory).

- The snapshots box provides buttons to take snapshots of the four other panels in png format. The file name of each snapshot is organised as follows:

  \[
  \text{program name}_msk\_\text{msk}\_\text{date}.png
  \]

  for the mask snapshot and

  \[
  \text{program name}_\text{iteration}_\text{panel name}_\text{options}_\text{date}.png
  \]

  for the others in which panel names are respectively gph, exp and str for the graph, explorer and structure panels. The iteration field is the iteration number at which the snapshot was taken. The explorer and structure files’ name can include some options separated by an underscore. For the explorer they are:

  - \text{v} followed by the view number,
  - \text{i} if interactions are displayed,
  - \text{b} if blinking is enabled,
  - \text{l} if labels are displayed.

For the structure they are:

  - \text{v} followed by the view number,
  - \text{u} if the system usage is represented,
  - \text{c} if the systems type changes are displayed,
  - \text{sk} if the skeleton is displayed,
  - \text{sy} if the systems are displayed,
  - \text{i} if the interactions are displayed,
  - \text{e} if the envelope is displayed,
– if labels are displayed.

The date checkbox adds to the file name the date and time up to the millisecond precision using the format YMD_hms_n where Y is the year, M the month, D the day, h the hours, m the minutes, s the seconds and n the milliseconds.

The snapshot files are put in the application root directory.

Mask panel

The masks panel allows to define templates (masks) matching some systems’ definition to give them a given colour. This enables a form of colour typing for various kinds of systems. By default the masks are set to the definitions of systems as given in the program code. The language used is the same as the systems’ definition (kernel and schemata).

For example the mask ??11 00?? ?? ?? would match any system with the ones and zeros located at the same place (e.g. 1111 0000 0000 or 00110010 1010) and give them the associated colour in all visualisations.

The tool bar on the left side allows, in order, to add a new mask, edit the selected mask, remove the selected mask, merge the selected masks, load masks from a mask file, save masks to a mask file, move the selected masks up and down.

Graph panel

The graph panel shows the hierarchy of the model using a graph that models systems as nodes with arrows between them going from super systems to sub systems. The right button of the mouse can be used to select a node (i.e. a system) or a group of nodes and drag it or them around. The left button can be used to place the explorer on the selected node. During computation the hierarchy of the model can change in which case the user may want a different organisation. The Rearrange button rearranges the graph layout automatically. Note that this discards the user’s changes to the graph layout.

Explorer panel

The explorer panel provides a 3D explorer view of the hierarchy. A physics engine is in charge of placing the systems so that they do not overlap. Systems are placed within their scopes. In this visualisation each system is represented once per scope, thus if a system has two super systems this very system is present once in each scope.

The user can browse through the hierarchy using the mouse. Starting from above the universe, it is possible to zoom in any system with the mouse left button and zoom out with the right button. The mouse wheel can be used similarly, forward to zoom in,
backward to zoom out. The graph panel can also be used to select the system to reach in the explorer. The camera can be rotated along the depth axis by pressing $S$ to rotate left or $D$ to rotate right.

- The *Physics* box allows to set up parameters of the physics engine used to position the systems.

  Scaling options are available:
  
  - The *Margin* slider sets the size of the empty space the engine should maintain between two systems.
  - The *Size* slider sets the size of the systems.

*Forces* exerted on the elements can be modified. When interactions occur the two interacting systems are temporarily linked to their context of interaction and attracted towards it.

  - The *Repulsion* force slider sets up the strength of the force pushing systems away from each other when getting too close (i.e. overlapping or within the margin distance).
  - The *Attraction* force slider sets up the strength of the attraction created by the interaction links.

*Systems* options are concerned with the systems’ bodies:

  - The *Reset Locations* button repositions systems randomly within their scopes.

- The *Display* box allows to set up parameters of the view. This does not affect the physics engine.

  - The *add* and *remove* view buttons respectively add a view to or remove the current view from the view tabs. By default the explorer is on *View 1* but more views can be used, each with its own camera location and display setup.
  - The camera *Speed* slider sets the camera motion speed and the button on the right resets the camera to its initial position.

The *Repaint* box sets the occurrence of the refreshing events:

  - *Never*: If rendering is not needed for a while but visual data is being recorded (e.g. to visualise a process at a later stage), this mode turns off the rendering and allows the computation to go faster.
  - *On Events*: Repaint is only done when a new computation occurs and impacts the visualisation. This mode is a trade-off between rendering and computation speed.
Appendix Platform Environment

- Always: This mode maintains a smooth and continuous motion that progressively places the elements after the occurrence of events.
- The button on the right sets the colour background.

The Show options enable or disable some visualisations.

- The Depth slider sets how many hierarchy levels should be recursively drawn.
- The Interactions checkbox shows or hides the interactions (i.e. links between systems and context).
- The TTL slider sets the time an interaction link remains visible before disappearing. Interaction links get transparent as they age until they disappear completely.
- The Inter. blink checkbox makes the interacting systems blink to greyscale one instant at the moment of the interaction.
- The Labels checkbox shows or hides the systems’ labels (i.e. names displayed). The slider sets up their font size.

Structure panel

The structure panel provides a 3D representation of the program as a structure where each system is represented once only by a sphere and connected to its super systems by a spring-like connection. All these hierarchy connections are called the skeleton. A physics engine is in charge of pushing the systems away from each other. One system is anchored in the space to prevent systems from drifting away. Systems linked to others are attracted by a force that maintains them closer than other systems they are not linked to. As all systems are contained directly or indirectly within the universe, there is no system that is not linked to any system (i.e. on its own). By default, the universe is the space anchor. When an interaction occurs, the two interacting systems are temporarily linked to their context of interaction for a certain amount of time. This connection is also a spring-like connection.

An abstract envelope can also be added onto the structure and reflects the overall computation (i.e. the past and the present). Full details about the structure and its envelope definition are given in Chapter 9.

The user can move through the space using the keyboard. The arrow keys shift the camera in the chosen direction. The shift key moves the camera forward and the control key moves it backward. The camera can be rotated along the depth axis by pressing S to rotate left and D to rotate right. The camera can look left by pressing X, look right by pressing C, look up by pressing F and look down by pressing V.
Appendix Platform Environment

• The Physics box allows to set up parameters of the physics engine used to position the systems.

  The Forces sliders provide control over the way physical forces are handled.

  – The Time Step slider sets the time instant duration which is the approximation of the mathematical differential expression of time $\delta t$. The longer this time value, the faster the motion but the coarser the approximation.

  – The Damping slider sets the percentage of the force that should be applied to the bodies. A high damping value makes the system settle faster but the settled position may not be as optimal as if using a smaller damping value.

  – The Threshold slider sets the amount of energy below which the system is considered as settled (i.e. no more motion is computed). The lower this value the more precise the final arrangement of systems, but the longer it takes to settle completely.

The Layout options provide the setup for the arrangement of systems.

  – The Anchor combo box sets the system that has a fixed location in the 3D space in order to keep the system centred and avoid it drifting away. It is by default set on the first system, usually the universe.

  – The Algorithm combo box allows to choose between two layout algorithms: the force-directed algorithm (FDA) that tries to distribute all systems away from each other (this is the default algorithm) and the hierarchy force-directed algorithm (HFDA) that only spaces out the systems part of a same scope.

  – The Repulsion force slider sets up the strength of the force pushing systems away from each other.

  – The Attraction force slider sets up the strength of the attraction created by the spring-like connections.

  – The Links TTL slider sets the life duration of the temporary interaction links.

  – The 2D/3D radio buttons indicate whether visualisation should be done on a 2D plan or extended to the 3D space. Visualisations in 2D are usually easier to understand but 3D visualisations offer more layout possibilities which can sometimes render clearer than in 2D.

The visualisation system has the possibility to keep track of the usage of systems over a certain amount of time (its setup is done in the Control panel). When systems are involved in a computation their usage goes up, and it goes down over time. As a result the more systems are used in computations, the higher their usage value. This value therefore reflects the current activity of systems and is
contained within the interval $[0,1]$. All systems are initialised with a usage value of $1$.

- The Usage on Z slider, only usable in 2D mode, renders the systems’ usage on the Z axis. The closer a system to the camera, the higher its usage value.

The systems in the structure have a form of electric charge. Each subsystem of a system is given a charge value that is the value of its super system’s multiplied by a factor.

- The Skeleton options’ Factor field allows to change this value. By default all charges are set to $1$. A stronger factor pushed system further away form each other, and a lower factor brings them closer.

In order to hide unnecessary information (i.e. some systems and their connections), make the visualisation clearer or speed up the process, some systems can be ignored in the rendering process.

- The Systems options’ Select button opens a dialog that allows to choose which systems are considered and which are not.
- The Reset Locations button repositions the systems randomly in the space.

To clear recorded data:

- The Interactions buttons deletes all records of interactions.
- The Envelope button deletes the current envelope data. The envelope is progressively built throughout computations, deleting its data allows to start a new one by discarding all past knowledge.

- The Display box allows to set up parameters of the view. This does not affect the physics engine.

- The add and remove view buttons respectively add a view to or remove the current view from the view tabs. By default the structure viewer is on View 1 but more views can be used, each with its own camera location and display setup.
- The camera Speed slider sets the camera motion speed and the button on the right resets the camera to its initial position.

The Envelope options set up the display of the structure’s abstract envelope. In the envelope, the more an interaction happens the wider its associated pipe in the envelope gets.

- The Min slider sets the minimum width of each pipe of the envelope.
– The *Amp* slider sets the amplitude of the envelope which is the maximum width the pipe representing the most common interaction would get.

– The *Opacity* slider sets the opacity of the envelope.

The *Repaint* box sets the occurrence of the refreshing events:

– *Never*: If rendering is not needed for a while but visual data is being recorded (e.g. to visualise a process at a later stage), this mode turns off the rendering and allows the computation to go faster.

– *On Events*: Repaint is only done when a new computation occurs and impacts the visualisation. This mode is a trade-off between rendering and computation speed.

– *Always*: This mode maintains a smooth and continuous motion that progressively places the elements after the occurrence of events.

– The button on the right sets the colour background.

The *Show* options enable or disable some visualisations.

– The *Usage* checkbox enables or disables the visualisation of systems usage. The usage is represented by transparency thus systems used frequently are opaque while the others are transparent and may disappear. The skeleton connections going to or coming from systems disappear as they do.

– The *Changes* checkbox enables or disables the tree-ring view of systems showing all their past types as rings within a half-sphere (systems’ top half-sphere is thus hidden) starting from the centre for the oldest types. The larger a ring, the longer the system remained in the associated type.

– The *Skeleton* checkbox shows or hides the skeleton. The button on the right allows to select the colour of the skeleton.

– The *Systems* checkbox shows or hides the systems. The *Select* button on the right allows to select the systems that should be shown. By default, the systems not considered by the physics engine are hidden.

– The *Interactions* checkbox shows or hides the interactions (i.e. links between systems and context). The button on the right allows to select the colour of the interaction links.

– The *Envelope* checkbox shows or hides the envelope.

– The *Labels* checkbox shows or hides the systems’ labels (i.e. names displayed). The slider sets up their font size.
A.5 Artificial Neural Network Application

The platform contains the application *SCANN* for the artificial neural networks model presented in Chapter 7, shown in Figure A.4. The aim of the application is to facilitate the work of the user by providing a GUI that enables easy experiment setup. The application is built on top of the virtual machine and is in charge of presenting the samples to the network (by writing on the universe), providing the value to learn (by writing on the universe) and collecting the outputs (by reading from the universe). It also displays on-line the response of the network to the presented samples.

Figure A.4: SCANN application
The various panels are detailed below:

- The *Program* panel from which a program can be loaded indicates the name of the program and its number of systems and contexts.

- The network representation panel provides a drawing of the network.

- The *Simulation* panel allows to:
  - Run or stop the computation using the *Run* toggle button,
  - Reset the program using the *Reset* button,
  - Pop up a window showing the systems using the *Show Systems* button.

- The *Data* panel contains two panels that respectively handle the sets and the samples.
  - The *Sets* panel allows to add, remove and clear datasets. The table gives for each set the average and the standard deviation of the values returned by the network over the set’s samples. The *Reset* button resets the classification results of all sets.
  - The *Set Samples* panel allows to add, remove and clear samples for the selected dataset. The value in the *Class* text box is the classification value to be learned by the network when given the samples.

The format of a sample file is:

* nb_samples value: the number of samples in the file
* type type: accepted values are float, integer, string or file
* size value: the size of each sample (i.e. how many inputs)
* data values: the sample values with each sample starting on a new line

Sample files for the X-Or truth table are given in Listing A.1 for the positive samples and in Listing A.2 for the negative samples.

- The output panel shows in *green* the classification value associated with the sample being presented to the network, and in *red* the actual response of the network over time. Once the network has learnt, the red line follows the green line, as can be seen on Figure A.4. The delay between the two lines is due to the time needed to process the sample through the network. When a new sample is given the expected output (green) line switches to the associated value but the response of the network changes only once the sample has been processed, which requires a few iterations.
• The Time setup panel sets:
  
  – The amount of iterations per sample: the amount of systemic computations allowed to process a sample before presenting the next sample,
  
  – The learning spike: the moment when the network receives the classification value to learn for the presented sample. The amount of iterations should allow enough time for a sample to be processed, at that moment the learning spike should be reached, then the same amount of iterations is necessary for the learning.

  How many iterations a network needs to process a sample depends on the network. Each iteration allows one interaction to happen. For a single neuron perceptron with two inputs, there are 6 contexts (2 input transfer, 1 output transfer, 2 synapse and 1 membrane context systems), the inputs need to be transferred to the axons, then the signal goes through the synapses, then through the membrane, then through the output transfer. Therefore the signal requires 4 successive steps to go through the network. Each successive step can be achieved in a cycle (see Section 4.2). Hence $4 \times 6 = 24$ iterations are needed to process a sample. For learning, 3 backpropagation steps are necessary (output transfer, membrane, synapses) as the learning does not involve the input transfer. $3 \times 6 = 18$ iterations are required. In total $24 + 28 = 42$ iterations are needed and the learning spike should be at 25.

• The Display setup panel enables to turn on or off the output display with the Update display checkbox. Turning it off allows to speed up computation. The Display buffer field sets the amount of memorised past iterations’ outputs.

• The Current state panel provides the amount of epochs and the amount of samples the network has been presented with, the currently presented set and sample.

• The Supervision panel sets the learning parameters. The amount of epochs used for learning can be set as well as the weight update mode, batch or on-line.

• The Process toggle button turns on or off the samples presentation to the network and the learning as set up in the previous panels. The button can be released in order to observe the response of a network for an indefinite amount of time, like for instance to observe a recursive network that has a non stable output.
LISTING A.1: Set of values classified as positive in the X-Or example

```plaintext
// Number of samples in the file
nb_samples 2

// Type of data {float, integer, string, file}
type integer

// Sample size (number of "type")
size 2

// Samples data
data
−1 1
1 −1
```

LISTING A.2: Set of values classified as negative in the X-Or example

```plaintext
// Number of samples in the file
nb_samples 2

// Type of data {float, integer, string, file}
type integer

// Sample size (number of "type")
size 2

// Samples data
data
−1 −1
1 1
```
Appendix B

SC Programming Reference

B.1 Introduction

To write a complete SC model, a description of the model’s systems, hierarchy and eligible interactions must be provided along with interaction function implementations, as described in Chapter 4. This appendix provides the complete reference to the SC language and to the C++ interface with the virtual machine via the plugins.

B.2 SC Language

This section lists all the keywords found in the SC language and the way to use them.

- //
  Single line comment symbol.
  Example:
  
  // Single line comment

- /* */
  Multiple lines comment notation.
  Example:
  
  /*
   Multiple lines
   comment
  */
• **import file_name**

Imports a file within the current file as if the imported file’s code was part of the current file’s.

Example:

```
// Imports file "file_name.sc" within current file
import "file_name.sc"
```


• **set name value**

Sets a variable with a numerical value. The variable can then be used anywhere instead of the numerical value.

Example:

```
set n 10
```

• **encoding**

Encoding section containing the compression code definition file, the list of plugins to load, the word length and the offset defining the range of values where to read the function code.

Example:

```
encoding {
   /* Content */
}
```

  – **char_map file**

Sets the file containing the compression code. This file should have two columns listing on the left the code and on the right its value as illustrated below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>000</td>
</tr>
<tr>
<td>a</td>
<td>001</td>
</tr>
<tr>
<td>b</td>
<td>00?</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The code needs to provide all the combinations for groups of three characters each being taken in 0,1,?. There are therefore $3^3 = 27$ combinations to provide in the code. By compressing three characters into one compressed character, a complete system definition which contains three words (i.e. two schemata and a kernel) can fit within one word (i.e. a schema).

Example:
encoding { 

    // Sets the file containing the compression code
    char_map "code_file.map"

    /* More content */

}

- func_libs { plugin_list }

Sets the plugin(s) associated with the current SC file.

Example:

encoding { 

    // Associates the given plugins to the current SC program
    func_libs { "plugin_1.dll", "plugin_2.dll" }

    /* More content */

}

- word_length value

Sets the word length which is the length of each schemata and the kernel. A full system definition is therefore three times the word length long.

Example:

encoding { 

    // Sets the word length to 256 characters
    word_length 256

    /* More content */

}

- func_offset first:last

Sets the range of characters that encode a function name, as defined when using the function keyword.

Example:
**label name string**

Creates a label referenced by its *name* with the given *string* value which can be a combination of 0, 1 and the wildcard ? and should have a length equal to the word length (defined using the *word_length* keyword within the *encoding* section). A label can use the operator < > to repeat a character a given amount of time. For a character c, it can be repeated n times by writing < c,n > or infinitely (until reaching the word length) by writing < c >. To fill in the space to reach the word length a label commonly ends with < ? >. A label can also contain characters from the compression code, if needed, but the manual use of compression code should be avoided and left to the compression operator.

Example:

```
label ANY <?>
label K_UNIVERSE <?,8> 0000 <?>
label SYSTEM <?,8> 0101 <?>
```

**function name string**

Creates a function label *string* containing a value that will be held in systems’ kernel and which role is to map to the given function *name* when the function is called. *string* should have a length equal to the word length. The function code part of the string should be a binary string, the rest should be wildcards. The part of the value where the code is located is defined using the *func_offset* keyword within the *encoding* section. To fill in the space to reach the word length a function label commonly ends with < ? >.

Example (using a function offset of 0:3):

```
function My_Function 0001 <?>
function My_Other_Function 0010 <?>
```
• **function NOP string**

Associates the NOP function with a binary value within label string, as defined above. In an SC program the NOP function label must be defined.

Example:

```
function NOP 0000 <?>
```

• **system name { schema1, kernel, schema2 }**

Defines a system giving it a name used to refer to its definition when instantiating systems in the program section. Both schema1, kernel and schema2 can be defined with a combination of labels using the operator |. Labels (having all a length equal to the word length) must not overlap with other values than the wildcard. If this case happens a compilation error will occur.

Example:

```
// Non-context definition
system My_System {
    ANY ,
    NOP | MY_SYSTEM ,
    ANY
}

// Context definition making two "My_System" systems interact
// as defined by the function "My_Function"
system My_Context {
    [ ANY, NOP | MY_SYSTEM, ANY ] ,
    My_Function | MY_CONTEXT ,
    [ ANY, NOP | MY_SYSTEM, ANY ]
}
```

• **program**

Program section containing the instantiations of systems and the initial systems hierarchy (the program then potentially modifying it during runtime). Instances are defined by providing a list of names after giving a system definition name (i.e. a name defined using the system keyword as explained above). Scopes are then provided using the brackets { and } to list, after a given system instance name, the systems that are held within it.

Example:
program {
    /* Instanciations */

    // Declares a universe
    Universe universe;
    // Declares two instances of "My_System"
    My_System system1, system2;
    // Declares an instance of "My_Context"
    My_Context context;

    /* Scopes */

    // Places instances "system1", "system2" and "context"
    // within instance "universe"
    universe { system1, system2, context }
}

• system [ index1:index2 ]

Array notation. It can be used to instantiates a system several times like an array in C++. The same notation can then be reused to refer to one or several instances when declaring the scopes.

Example:

program {
    /* Instanciations */

    // Declares ten instances of "MySystem" with an array notation
    // using indices 1 to 10.
    MySystem my_system[1:10];

    /* Scopes */

    // Puts my_system instances 2 to 10 within the first one
    my_system[1] { my_system[2:10] }
    // Loop that puts instances 3 and 6 within instances 2 to 4
    // The last line is equivalent to the three following
    // commented lines together:
}
// and also to the following pseudo code:
// for i = 2 to 4 do
//   my_system[i] { my_system[3], my_system[6] }

• idx

At scope declaration, when using the loop array notation to declare scopes for several systems in one line, the index of the current scope being processed can be accessed using the idx keyword.

Example:

```plaintext
program {

/* Instanciations */
Universe universe;
MySystem my_system[1:10];
MySystem my_sub_system[1:10];

/* Scopes */

// Puts all "my_system" system within the universe
universe { my_system[1:10] } // Puts within each "my_system" the corresponding system having the same index.
my_system[1:10] { my_sub_system[idx] } // This is equivalent to the pseudo code:
// for i = 1 to 10 do
//   my_system[i] { my_sub_system[i] }

}
B.3 C++ Platform Library Interface

Note: The following appendix requires a minimum understanding of the C++ language and knowledge of its standard library as well as the STL (Standard Template Library) library. http://www.cppreference.com and http://www.cplusplus.com both provide a complete reference for C++ and STL language and library. The latter also provides a complete language tutorial.

While an SC language program provides a complete description of models’ organisation, the code of the interaction functions performing the transformation of the interacting systems has to be written in C++ within a plugin associated with the SC program defining the model. These functions are called and executed by the virtual machine in order to modify the state of its systems. The systems are given to the function as parameters using C++ references to the system objects held by the virtual machine. Therefore system objects are handled using the same system interface the virtual machine uses internally.

Each interaction function has a prototype as shown in Listing B.1 and usually given in a plugin header file.

Listing B.1: Interaction function prototype

```cpp
bool function(SC::SCSystem &, const SC::SCSystem &, SC::SCSystem &);
```

The given arguments are three system objects: the first interacting system, the context of interaction and the second interacting system. Note that the word `const` is not mandatory. The plugin’s header file must include beforehand the library header `SCSystem.hpp` which provides the interface of the `SCSystem` object, as shown in Listing B.2. The `SCSystem` object is part of the namespace `SC`, hence the prefix `SC::` indicating where to find the object’s declaration.

Listing B.2: Inclusion of the SCSystem header file

```cpp
#include "SCSystem.hpp"
```

Each plugin also has two functions that are respectively called at loading and closing. Their role is mainly to properly initialise and free memory or open and close files that might be used for data recording when running a model. Listing B.3 provides the interface of these functions placed in the header of the plugin. They are called if they exist and do not have to be implemented if not needed.
Listing B.3: Plugin initialisation and exit functions

```cpp
void InitPlugin (const std::string &);

void ClosePlugin (void);
```

The `string` parameter is an STL string object that gives the name of the model being simulated. Several models can use a same plugin to for instance create a log file that is associated with a given model, having the current model’s name is then convenient to properly name the log file. The plugin’s header file should include the C++ string library header `string` and `string` objects should be prefixed by `std::`.

The header of a plugin thus has the structure given in Listing B.4.

Listing B.4: Structure of a plugin’s header

```cpp
#include "SCSystem.hpp"
#include <string>

void InitPlugin (const std::string &);

void ClosePlugin (void);

bool Function1(SC::SCSystem &, const SC::SCSystem &, SC::SCSystem &);

bool Function2(SC::SCSystem &, const SC::SCSystem &, SC::SCSystem &);
```

The implementation of these functions can then be provided in a separate “.cpp” file. (Some programmers might prefer to put all the code also within the header though). The methods available publicly for the `SCSystem` object are provided in Listing B.5.

Listing B.5: SCSystem public methods

```cpp
/// Grabs the given system
const bool Grab(SCSystem &);

/// Takes the ownership of the given system from the shared
/// super-systems between the two systems interacting and the context
const bool TakeFromSuperSystems(SCSystem &, const SCSystem &);

/// Releases the given system
const bool Release(SCSystem &);

/// Releases the given system to the current system’s parents
```
const bool ReleaseToSuperSystems(SCSystem &);

/// Returns the required substring from the first schema
const std::string GetSchema1(
    const unsigned int = 0, const unsigned int = 0
) const;

/// Replaces the desired substring in the first schema
void ReplaceSchema1(
    const std::string &, const unsigned int = 0, const unsigned int = 0
);

/// Returns the required substring from the kernel
const std::string GetKernel(
    const unsigned int = 0, const unsigned int = 0
) const;

/// Replaces the desired substring in the kernel
void ReplaceKernel(
    const std::string &, const unsigned int = 0, const unsigned int = 0
);

/// Returns the required substring from the second schema
const std::string GetSchema2(
    const unsigned int = 0, const unsigned int = 0
) const;

/// Replaces the desired substring in the second schema
void ReplaceSchema2(
    const std::string &, const unsigned int = 0, const unsigned int = 0
);

/// Returns the total definition
const std::string GetDefinition() const;

/// Returns the number of super systems
const unsigned int NbSuperSystems() const;

/// Returns the ith super system (read-only)
const SCSystem & GetSuperSystem(const unsigned int) const;

/// Returns the number of sub systems
const unsigned int NbSubSystems() const;
/// Returns the ith sub system (read-only)
const SCSystem & GetSubSystem(const unsigned int) const;

/// True if the given system is a super system of the current one
const bool HasSuperSystem(const SCSystem &) const;

/// True if the given system is a sub system of the current one
const bool HasSubSystem(const SCSystem &) const;

/// Returns the systems shared with the given one
const std::vector<const SCSystem *> ShareWith(const SCSystem &) const;

/// Returns true if the systems share another one
const bool DoesShareWith(const SCSystem &) const;
References


Mark Buchanan. *Ubiquity: The science of history... or why the world is simpler than we think.* Phoenix, 2000.


REFERENCES


George P. Conger. The doctrine of levels. The Journal of Philosophy, 22(12):309–321, 1925. ISSN 0022362X.


REFERENCES


Elisabeth Gasteiger, Alexandre Gattiker, Christine Hoogland, Ivan Ivanyi, Ron D. Appel, and Amos Bairoch. Expasy: the proteomics server for in-depth protein


REFERENCES


REFERENCES


Edward N. Lorenz. Predictability: Does the flap of a butterfly’s wings in brazil set off a tornado in texas? In *139th Meeting of the American Association for the Advancement of Science*, 1972.


Benoît Mandelbrot. Fractal aspects of the iteration of $z \rightarrow \lambda z(1- z)$ for complex $\lambda$ and $z$. *New York Academy Sciences Annals*, 357:249–259, December 1980.


REFERENCES


References


REFERENCES


Mark Ward. There’s an ant in my phone... *New Scientist*, (2118), 1998.


REFERENCES


