Evolvable Virtual Machines

Mariusz Nowostawski

a thesis submitted for the degree of
Doctor of Philosophy
at the University of Otago, Dunedin,
New Zealand.

December 2007
Abstract

The Evolvable Virtual Machine abstract architecture (EVMA) is a computational architecture for dynamic hierarchically organised virtual machines. The concrete EVM instantiation (EVMI) builds on traditional stack-based models of computation and extends them by notions of hierarchy and reflection on the virtual machine level. The EVM Universe is composed of a number of autonomous and asynchronously communicating EVM machines. The main contribution of this work lies in the new model of computation and in the architecture itself: a novel, compact, flexible and expressive representation of distributed concurrent computation. The EVMA provides a way of expressing and modelling auto-catalytic networks composed of a hierarchical hypercycle of autopoietic subsystems characterised by self-adaptable structural tendencies and self-organised criticality. EVMA provides capabilities for: a) self-learning of dynamical patterns through continuous observation of computable environments, b) self-compacting and generalisation of existing program structures, c) emergence of efficient and robust communication code through appropriate machine assembly on both ends of communication channel. EVMA is in one sense a multi-dimensional generalisation of stack machine with the purpose of modelling concurrent asynchronous processing. EVMA approach can be also seen as a meta-evolutionary theory of evolution. The EVMA is designed to model systems that mimic living autonomous and adaptable computational processes. The EVMI prototype has been designed and developed to conduct experimental studies on complex evolving systems. The generality of our approach not only provides the means to experiment with complex hierarchical, computational and evolutionary systems, but it provides a useful model to evaluate, share and discuss the complex hierarchical systems in general. The EVMA provides a novel methodology and language to pursue research, to understand and to talk about evolution of complexity in living systems. In this thesis, we present the simple single-cell EVMI framework, discuss the multi-cell EVM Universe architecture, present experimental results, and propose further extensions, experimental studies, and possible hardware implementations of the EVMI.
Acknowledgements

Many people helped with professional and personal support to make this work possible. My sincere gratitude to my wife, Urszula Nowostawska, for patience and her unconditional love. My daughter Emilia, for showing me and teaching me the miracle and innocence of life itself. My parents Daniela and Józef, my brother Michal and sister Izabela. My extended family, my friends, among others: Marcos Oliveira, Tony Moore, Tony and Sharmila Savarimuthu, Carl Leichter, Lucien Epiney, Mengqiu Wang, Rumen Raykov, QingQing Zhou, Da Deng, Paweł Waszkiewicz, Andrzej Gecow, and many others, who have, professionally and personally, helped this project to progress and to achieve its current status.

Big thanks to my friend and mentor, Martin Purvis for numerous long discussions, suggestions and critical evaluation of this work. I thank Stephen Cranefield, Noria Foukia and Tony Moore for an objective, critical and thorough evaluation of a draft of this thesis. I would also like to thank anonymous reviewers of multiple publications related to this work. Many thanks to Carl Leichter, William B. Langdon, Roland Olsson, and Roy Ward for help, feedback and discussions on publication on different topics related to this work.

Anti-acknowledgements. I would like also to thank friends, who made my PhD journey longer than necessary, by providing multiple distractions and directing my attention away from the research and scientific pursuits. Thanks to them I have experienced the taste of adventure, mountaineering, scuba diving, rock climbing and paragliding. These people made my life worthwhile. We have enjoyed multiple trips into New Zealand wilderness and lots of great time together. Many thanks to: David Winzler, Jay Piggot, Lucien Epiney, Mengqiu Wang, Sonja Schaper, Coral Conant and others. Also, many thanks to members of the Otago University Diving Club, Polish Heritage of Otago and Southland, Otago University Tramping Club and Dunedin Photographic Society. Thank you all.
To whom I love
# Contents

1 Introduction .......................... 5
  1.1 Motivation .......................... 7
  1.2 Modelling, scope and objectives ........................................ 9
    1.2.1 Technological advances ........................................... 10
    1.2.2 Meta-evolution ................................................... 11
    1.2.3 Theory of computation .......................................... 12
    1.2.4 Philosophical objectives ....................................... 14
  1.3 The thesis ............................ 15
  1.4 Current research within life sciences ............................... 16
  1.5 Organisation of this work .......................................... 19
  1.6 Summary of the contribution ....................................... 20

2 EVM Abstract Architecture ................. 23
  2.1 Introduction .......................... 24
  2.2 General ideas and inspiration ..................................... 24
  2.3 Virtual machine ................................................. 27
    2.3.1 Short history of the concept .................................. 28
  2.4 Design principles ............................................... 29
  2.5 The EVM abstract architecture ................................... 30
    2.5.1 Introduction .................................................. 30
    2.5.2 Abstraction ................................................... 30
    2.5.3 Levels ......................................................... 30
    2.5.4 Expressiveness ............................................... 31
    2.5.5 Incompleteness ............................................... 32
    2.5.6 Hierarchy and cross-level linking ............................. 33
    2.5.7 Constructions within the system ................................ 34
  2.6 Tasks ....................................... 35
  2.7 Cells ........................................ 36
    2.7.1 Intra-cell computing ....................................... 36
    2.7.2 Inter-cell computing ....................................... 36
  2.8 Environment and cell topology ......................... 37
  2.9 Complexity measure ........................................ 38

3 EVM Instantiation .......................... 41
  3.1 Introduction ...................................... 42
  3.2 Bootstrapping implementation ................................. 42
  3.3 Tasks ........................................ 43
  3.4 Cells ........................................ 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 The Computable Universe</td>
<td>98</td>
</tr>
<tr>
<td>4.5 Supertasks</td>
<td>101</td>
</tr>
<tr>
<td>4.6 Hypercomputation</td>
<td>103</td>
</tr>
<tr>
<td>4.6.1 Is hypercomputing possible?</td>
<td>104</td>
</tr>
<tr>
<td>4.7 Massively parallel asynchronous computation</td>
<td>105</td>
</tr>
<tr>
<td>4.7.1 Preliminaries and motivation</td>
<td>105</td>
</tr>
<tr>
<td>4.7.2 Interactive machines</td>
<td>105</td>
</tr>
<tr>
<td>4.7.3 Persistent Turing machines</td>
<td>106</td>
</tr>
<tr>
<td>4.7.4 Multiple Interactive Machines</td>
<td>107</td>
</tr>
<tr>
<td>4.7.5 Algorithmic hierarchy</td>
<td>107</td>
</tr>
<tr>
<td>4.7.6 Interactive hierarchy</td>
<td>109</td>
</tr>
<tr>
<td>4.8 Generalised computation</td>
<td>110</td>
</tr>
<tr>
<td>4.8.1 Algorithmic abstractions</td>
<td>110</td>
</tr>
<tr>
<td>4.8.2 Computing machines</td>
<td>111</td>
</tr>
<tr>
<td>4.8.3 Realisation of computing machines</td>
<td>112</td>
</tr>
<tr>
<td>4.8.4 Algorithmic complexity</td>
<td>112</td>
</tr>
<tr>
<td>4.8.5 The No Free Lunch Theorem</td>
<td>113</td>
</tr>
<tr>
<td>4.9 Hierarchical Computation</td>
<td>114</td>
</tr>
<tr>
<td>4.9.1 Levels of organisation</td>
<td>115</td>
</tr>
<tr>
<td>4.9.2 Interpretations and organising relations</td>
<td>115</td>
</tr>
<tr>
<td>4.10 Virtual machines</td>
<td>117</td>
</tr>
<tr>
<td>4.10.1 Motivation</td>
<td>117</td>
</tr>
<tr>
<td>4.10.2 Formal definitions</td>
<td>117</td>
</tr>
<tr>
<td>4.10.3 Modelling hierarchies</td>
<td>118</td>
</tr>
<tr>
<td>4.10.4 Vertical and horizontal decomposition</td>
<td>119</td>
</tr>
<tr>
<td>4.10.5 Partial equivalence</td>
<td>121</td>
</tr>
<tr>
<td>4.10.6 Decomposition limits</td>
<td>122</td>
</tr>
<tr>
<td>4.11 Formal EVMA extensions</td>
<td>123</td>
</tr>
<tr>
<td>4.11.1 Equivalence classes</td>
<td>123</td>
</tr>
<tr>
<td>4.11.2 Beyond algorithmic function composition</td>
<td>123</td>
</tr>
<tr>
<td>4.11.3 Generalised EVMA computation</td>
<td>125</td>
</tr>
<tr>
<td>4.12 Summary</td>
<td>125</td>
</tr>
<tr>
<td>5 Theory of Evolution</td>
<td>127</td>
</tr>
<tr>
<td>5.1 Theory of Evolution</td>
<td>128</td>
</tr>
<tr>
<td>5.1.1 Answers that we seek</td>
<td>128</td>
</tr>
<tr>
<td>5.1.2 Pre-Darwinian formulations</td>
<td>129</td>
</tr>
<tr>
<td>5.1.3 Darwinism</td>
<td>132</td>
</tr>
<tr>
<td>5.1.4 Modern synthesis, neo-Darwinism</td>
<td>133</td>
</tr>
<tr>
<td>5.2 Neo-Darwinism</td>
<td>135</td>
</tr>
<tr>
<td>5.2.1 DNA and RNA</td>
<td>135</td>
</tr>
<tr>
<td>5.2.2 Exons and introns</td>
<td>137</td>
</tr>
<tr>
<td>5.2.3 Orthodox gene-centred view</td>
<td>137</td>
</tr>
<tr>
<td>5.2.4 The search for a unifying theory</td>
<td>138</td>
</tr>
<tr>
<td>5.3 Beyond neo-Darwinism</td>
<td>138</td>
</tr>
<tr>
<td>5.3.1 The RNA world</td>
<td>139</td>
</tr>
<tr>
<td>5.3.2 Exaptation</td>
<td>139</td>
</tr>
<tr>
<td>5.3.3 Symbiosis</td>
<td>141</td>
</tr>
<tr>
<td>5.3.4 Cooperation</td>
<td>142</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Symbiogenesis</td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Epigenetics</td>
</tr>
<tr>
<td>5.3.7</td>
<td>Rnomics</td>
</tr>
<tr>
<td>5.3.8</td>
<td>Specialisation</td>
</tr>
<tr>
<td>5.4</td>
<td>Stem cells</td>
</tr>
<tr>
<td>5.5</td>
<td>The Baldwin effect</td>
</tr>
<tr>
<td>5.6</td>
<td>Neutral evolution</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Genetic drift</td>
</tr>
<tr>
<td>5.6.2</td>
<td>The neutral theory of molecular evolution</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Neutral computational evolution</td>
</tr>
<tr>
<td>5.7</td>
<td>The EVM biological inspirations</td>
</tr>
<tr>
<td>6</td>
<td>Artificial Evolution</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Artificial evolution and artificial life</td>
</tr>
<tr>
<td>6.3</td>
<td>Automata models</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Self-replication</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Growth models</td>
</tr>
<tr>
<td>6.4</td>
<td>Evolutionary computation</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Genetic Algorithms</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Genetic Programming</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Grammatical Evolution</td>
</tr>
<tr>
<td>6.5</td>
<td>Beyond neo-Darwinism</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Modelling adaptation</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Evolution without selection</td>
</tr>
<tr>
<td>6.6</td>
<td>Meta-evolution: going beyond traditional models</td>
</tr>
<tr>
<td>6.6.1</td>
<td>ADATE</td>
</tr>
<tr>
<td>6.6.2</td>
<td>PushPop</td>
</tr>
<tr>
<td>6.7</td>
<td>Artificial life and Artificial chemistry</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Typogenetics</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Binary reaction networks</td>
</tr>
<tr>
<td>6.7.3</td>
<td>Tierra and Avida</td>
</tr>
<tr>
<td>6.8</td>
<td>Evolvability</td>
</tr>
<tr>
<td>6.9</td>
<td>Hierarchy in Evolutionary Computation</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Evolving recursive virtual machines</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Seeds and solution growing</td>
</tr>
<tr>
<td>6.9.3</td>
<td>n-XOR problem</td>
</tr>
<tr>
<td>6.9.4</td>
<td>n-XOR hierarchy</td>
</tr>
<tr>
<td>6.10</td>
<td>The EVMA as an ALife model</td>
</tr>
<tr>
<td>6.11</td>
<td>Comparison with Swarm and RePast systems</td>
</tr>
<tr>
<td>7</td>
<td>Autopoietic hypercycles</td>
</tr>
<tr>
<td>7.1</td>
<td>Computational evolution</td>
</tr>
<tr>
<td>7.2</td>
<td>Information-centred models of evolution</td>
</tr>
<tr>
<td>7.3</td>
<td>Properties of computational evolution</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Split and splice</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Cyclic behaviour</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Many agents on many levels</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Hypothesis testing</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>7.3.5</td>
<td>External information flow</td>
</tr>
<tr>
<td>7.3.6</td>
<td>The sustainability condition</td>
</tr>
<tr>
<td>7.4</td>
<td>Darwinian systems</td>
</tr>
<tr>
<td>7.5</td>
<td>Hypercycles</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Definition of the hypercycle</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Hierarchies</td>
</tr>
<tr>
<td>7.6</td>
<td>Autopoiesis</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Machines</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Definitions</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Identity</td>
</tr>
<tr>
<td>7.6.4</td>
<td>Hierarchies</td>
</tr>
<tr>
<td>7.6.5</td>
<td>EVM and the theory of autopoiesis</td>
</tr>
<tr>
<td>7.7</td>
<td>Artificial and Natural life</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Causality and relativism of life</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Spontaneous autopoietic hypercycle</td>
</tr>
<tr>
<td>8</td>
<td>Learning how to learn</td>
</tr>
<tr>
<td>8.1</td>
<td>Biological learning</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Early theories of learning</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Law of Effect</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Short term vs. long term reinforcers</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Multi-task learning in biology</td>
</tr>
<tr>
<td>8.1.5</td>
<td>The pieces of the puzzle</td>
</tr>
<tr>
<td>8.2</td>
<td>Machine learning</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Early reinforcement learning</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Life-long reinforcement learning</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Multi-task systems</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Meta-learning</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Applications and advantages</td>
</tr>
<tr>
<td>8.3</td>
<td>Self-learning system</td>
</tr>
<tr>
<td>8.4</td>
<td>Environment-Independent Reinforcement Acceleration – EIRA</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Reinforcement Acceleration Criterion (RAC)</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Some comments about EIRA</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Implementation of policies</td>
</tr>
<tr>
<td>9</td>
<td>Experimental studies</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.2</td>
<td>Film analysis vs. quantitative studies</td>
</tr>
<tr>
<td>9.3</td>
<td>A-grid: pre-EVMI model</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Description</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Experiments</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Discussion of the results</td>
</tr>
<tr>
<td>9.4</td>
<td>Symbolic regression: preliminary studies</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Description</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Experimental setup</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Results and discussion</td>
</tr>
<tr>
<td>9.5</td>
<td>Machine discovery</td>
</tr>
<tr>
<td>9.6</td>
<td>Virtual ecosystem</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Description</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>9.6.2</td>
<td>Cellular topology</td>
</tr>
<tr>
<td>9.6.3</td>
<td>Complexity of tasks</td>
</tr>
<tr>
<td>9.6.4</td>
<td>Program discovery within a cell</td>
</tr>
<tr>
<td>9.6.5</td>
<td>Exhaustive search</td>
</tr>
<tr>
<td>9.7</td>
<td>Cell specialisation mechanisms</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Objectives</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Maze experiments</td>
</tr>
<tr>
<td>9.7.3</td>
<td>Arithmetical experiments</td>
</tr>
<tr>
<td>9.8</td>
<td>Random Search mechanism</td>
</tr>
<tr>
<td>9.8.1</td>
<td>Description</td>
</tr>
<tr>
<td>9.8.2</td>
<td>Observations</td>
</tr>
<tr>
<td>9.9</td>
<td>Stochastic Search mechanism</td>
</tr>
<tr>
<td>9.9.1</td>
<td>Description</td>
</tr>
<tr>
<td>9.9.2</td>
<td>Management of probabilities</td>
</tr>
<tr>
<td>9.9.3</td>
<td>Observations</td>
</tr>
<tr>
<td>9.10</td>
<td>Tree mechanism</td>
</tr>
<tr>
<td>9.10.1</td>
<td>Objectives</td>
</tr>
<tr>
<td>9.10.2</td>
<td>Conditional probabilities</td>
</tr>
<tr>
<td>9.10.3</td>
<td>Storing patterns</td>
</tr>
<tr>
<td>9.10.4</td>
<td>Reward mechanism</td>
</tr>
<tr>
<td>9.10.5</td>
<td>Selecting a program</td>
</tr>
<tr>
<td>9.10.6</td>
<td>Additional mechanisms</td>
</tr>
<tr>
<td>9.11</td>
<td>Summary of experiments</td>
</tr>
<tr>
<td>10</td>
<td>Summary</td>
</tr>
<tr>
<td>10.1</td>
<td>Summary and Comments</td>
</tr>
<tr>
<td>10.2</td>
<td>The EVM as a machine learning system</td>
</tr>
<tr>
<td>10.3</td>
<td>The EVM as an artificial life simulator</td>
</tr>
<tr>
<td>10.4</td>
<td>The EVM as a cellular system</td>
</tr>
<tr>
<td>10.5</td>
<td>Computational limitations</td>
</tr>
<tr>
<td>10.6</td>
<td>Research issues and future work</td>
</tr>
<tr>
<td>10.7</td>
<td>Complex interaction networks</td>
</tr>
<tr>
<td>10.7.1</td>
<td>The EVMA and the Internet</td>
</tr>
<tr>
<td>10.7.2</td>
<td>Application to genetic networks</td>
</tr>
<tr>
<td>10.7.3</td>
<td>Reflection</td>
</tr>
<tr>
<td>A</td>
<td>Notes on information theory</td>
</tr>
<tr>
<td>A.1</td>
<td>Measures</td>
</tr>
<tr>
<td>A.2</td>
<td>Code and encoding</td>
</tr>
<tr>
<td>A.3</td>
<td>Objects</td>
</tr>
<tr>
<td>A.4</td>
<td>Reversed code</td>
</tr>
<tr>
<td>A.5</td>
<td>Naturalistic interpretation</td>
</tr>
<tr>
<td>A.6</td>
<td>Theory of life and the notion of purpose</td>
</tr>
<tr>
<td>A.7</td>
<td>Complexity</td>
</tr>
</tbody>
</table>
B Biological life and the EVMA

B.1 Defining life .................................................. 283
B.2 Testing for life ............................................... 284
B.3 Symmetries in the definitions ............................. 285
B.4 Life within life .............................................. 286
B.5 Communication ............................................. 286
B.6 Looking ahead ............................................... 288

C The EVM’s universe and the Universe

C.1 Introduction .................................................. 289
C.2 Hypercomputation ......................................... 291
C.3 Evolvable Virtual Machines ............................... 293
C.4 Computational Universe .................................. 294
C.5 The need for reflection .................................... 295
C.6 Physics and new computationalism ..................... 296
C.7 Randomness .................................................. 298
C.8 Observations .................................................. 298
C.9 Summary ....................................................... 299
List of Tables

1 Glossary and acronyms used ......................................................... 4
3.1 Testing environments. ................................................................. 76
3.2 Performance results: time summary. ............................................. 88
3.3 Performance results: implementation speedup. ............................ 88
3.4 Performance results: speedup between CPU architectures. ............. 89
5.1 Example of cell specialisation. ..................................................... 149
6.1 Apter’s growth model, for a line of 2 automata. ............................ 160
6.2 Apter’s growth model, for line of 3 automata. ............................... 161
6.3 Apter’s growth model, for line of 5 automata. ............................... 162
6.4 Apter’s growth model, X-shaped 5 automata. ............................... 162
6.5 Apter’s growth model, Y-shaped 5 automata. ............................... 163
9.1 Some good solutions for maze experiments. ................................. 249
9.2 Some examples of the arithmetical tasks tackled. ......................... 251
9.3 Parameters for the random search specialisation mechanism. ............. 252
9.4 Parameters for the stochastic search specialisation mechanism. .......... 260
9.5 Subsequence example. ............................................................... 262
9.6 Parameters for the tree-based specialisation mechanism. .................. 266
List of Figures

3.1 Reusing neighbour programs – schema. ........................................ 49
3.2 Schematic view of EVM hierarchy. ............................................. 50
3.3 Reusing neighbour programs – example. ...................................... 51
3.4 Screenshot of the EVM IDE. .................................................... 53
3.5 EVM Execution Frames. ......................................................... 70
4.1 Schematic view of a Turing machine. ....................................... 97
4.2 Hierarchy of machines based on the Chomsky Hierarchy. ............. 109
4.3 No-free-lunch theorem. ......................................................... 113
4.4 A vertical split of machine M into a tower of machines M0, M1, M2. 119
4.5 A horizontal split of machine M into machines M0, M1, M2. .......... 120
4.6 Example of possible dependencies between machines. ............... 121
5.1 Space filling model of a section of a DNA molecule. ................... 135
5.2 Schematic representation of the chemical structure of DNA. ........ 136
5.3 Mereschkowsky’s symbiosis. .................................................. 144
5.4 Examples of cell specialisation. ............................................... 150
6.1 An example view of Conway’s Game of Life cellular automata. .... 158
6.2 Growth models. Apter model for growing a line of 2 automata. .... 161
7.1 Three-members catalytic cycle. ............................................... 194
7.2 Schematic view of hypercycle. ............................................... 194
8.1 Meta-instructions and system policies. ..................................... 216
8.2 System policies and probability distributions. ........................... 219
9.1 A-Grid missing links. .......................................................... 226
9.2 A-Grid established links. ...................................................... 228
9.3 A-Grid: cluster (length 100). ................................................. 229
9.4 A-Grid: hypercycle, (length 3). ............................................. 231
9.5 A-Grid: hypercycle, (length 100). .......................................... 231
9.6 A-Grid: initial linking (A). ................................................... 232
9.7 A-Grid: initial linking (B). ................................................... 232
9.8 Toroidal space. ............................................................... 240
9.9 Typical evolution of a cellular system. ..................................... 242
9.10 Internal execution within a cell. .......................................... 245
9.11 Example grid experiments. ................................................... 248
9.12 Self-assembly. ............................................................... 250
9.13 The random search mechanism. .......................................... 251
9.14 Probability distribution of the instruction. .......................... 254
9.15 Evolution of probability distributions. .................................. 256
9.16 Stochastic search and probabilities. ..................................... 257
9.17 The freezing mechanism. .................................................. 258
9.18 Cell and instruction freezing mechanisms. .............................. 258
9.19 Instruction sequences and storage of patterns. ....................... 261
9.20 Reward mechanism example. ............................................. 262
9.21 Program selection example. .............................................. 264
9.22 Freezing in the tree structure. ........................................... 265
My eyes seek reality

*Low man’s Lyrics*, Re-Load, Metallica (1997)
Credits

This study is a result of collaborative efforts of various people throughout various stages of the project. This thesis is a collection of different research ideas of many people, that has been compiled, linked and presented in this particular form. I have accessed and re-used ideas of others mostly through their writing, but also through personal interactions and (often long) discussions. The computations that have been performed, with a pen and a paper (or rather mostly in my head alone) is a result of what could be referred as hypercomputation\(^1\), that is computation that goes beyond the Turing model. I am certain that as such, this computation is unique and has not been performed before. However, I cannot claim, that this work is exclusively mine. This work would not be possible without all the input from other researchers and thinkers I have received and used throughout the process.

I tried hard to pay credit were it is due and to acknowledge all the people and theories that came before and those that my work is based on. During the process however, inherently, some ideas I have learned became so entrenched into my own way of thinking, that it becomes impossible to distinguish what have been thought by me alone and what have been learned/read earlier from others. I do not think such a distinction is of any significance apart from social conventions. After awhile, the theory and particular thought framework becomes a uniform and consistent network of inter-related ideas, some of which may have been mine, and some thought of by others earlier, even without me realising it.

And where do I get all these facts [...]. In [a] certain sense, from my head; in another, not. I have invented several picturesque similes to illustrate for myself and others what my working method is like:

(1) A cow produces milk – that is certain – and the milk doesn’t come from nothing. Just as a cow must eat grass in order to be able to produce milk, I have to read large amounts of genuine scientific literature of all kinds – i.e., literature not invented by me – and the final product, my writing, is as unlike the intellectual food as milk is unlike grass.

(2) Just as the ape in Wolfgang Köhler’s psychological experiments wasn’t able to reach a banana hanging very high, and made a scaffold from junk-boxes lying around, etc. – in order to be able to climb up to the banana, I have to build up, in subsequent moves and attempts an informational “scaffold” that I must climb up to reach my goal.

Lem (1984)

Many people helped me in shaping my own mind and ideas directly through discussions and direct interactions. The most important role was played by my supervisor, Prof. Martin Purvis, and other collaborators and colleagues from the Information Science Department, including Stephen Cranefield, Noria Foukia, Tony Moore, Bastin Tony Roy Savarimuthu, Marcos de Oliveira, and others.

\(^1\)See Chapter 4, Section 4.6 onwards
Many anonymous reviewers provided helpful comments to multiple articles and book chapters written throughout the course of my studies. I supervised and collaborated with several students who provided valuable feedback and help. Lucien Epiney provided substantial help with the fixed grid experiments. Based on our collaborative work I have published with Lucien several research articles presenting our research. I have been allowed to re-use some of Lucien’s excellent figures and graphics throughout this thesis. Thanks to his help, the visual presentation of the material related to fixed grid experiments is now greatly enhanced. I have used excerpts from publications related to the EVM architecture that have been published during the course of my studies. In addition I have also used definitions and examples from many web resources, most notably: http://www.biology-online.org, http://www.biology.arizona.edu, and http://www.wikipedia.org.

It has been challenging to present this material in a way that is interesting and approachable.

- The research and the notions related to massively parallel asynchronous systems cover broad range of research areas and a wide spectrum of different ideas, terminology and models. It is difficult to link them all into a complex web of interdependencies and relationships.

- It is difficult to propose a new way of looking at certain well-established and entrenched models in traditional fields of research in such a way that the message gets through without interfering with the reader’s own perspectives and world-views. This is most noticeable with the EVM architecture considered as a meta-evolutionary modelling framework.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALife</td>
<td>Artificial Life</td>
</tr>
<tr>
<td>CA</td>
<td>Cellular Automata</td>
</tr>
<tr>
<td>CMT</td>
<td>Chip-level Multi-threading</td>
</tr>
<tr>
<td>EC</td>
<td>Evolutionary Computation</td>
</tr>
<tr>
<td>EVM</td>
<td>Evolvable Virtual Machine</td>
</tr>
<tr>
<td>EVMA</td>
<td>Evolvable Virtual Machine abstract Architecture</td>
</tr>
<tr>
<td>EVMI</td>
<td>Evolvable Virtual Machine concrete Instantiation</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithms</td>
</tr>
<tr>
<td>GP</td>
<td>Genetic Programming</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>M</td>
<td>Machine</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>P</td>
<td>Program</td>
</tr>
<tr>
<td>TM</td>
<td>Turing Machine</td>
</tr>
<tr>
<td>UTM</td>
<td>Universal Turing Machine</td>
</tr>
<tr>
<td>UVM</td>
<td>Universal Virtual Machine</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>

Table 1: Glossary and acronyms used
Chapter 1

Introduction

No, I don’t like work. I had rather laze about and think of all the fine things that can be done. I don’t like work – no man does – but I like what is in the work, – the chance to find yourself. Your own reality – for yourself, not for others – what no other man can ever know. They can only see the mere show, and never can tell what it really means.

Conrad (1926, p.85)

Foreword

This thesis contains a summary of studies of a novel computational architecture, the Evolvable Virtual Machines abstract architecture (EVMA), which grew out of interests in and research into modelling meta-evolution. A novel methodology and framework is described here for modelling computational evolutionary processes, adaptation and computational properties of the process of artificial (and to a certain degree biological) life. The main objective and motivation to undertake this study was to find answers to many existential questions, such as the nature of life, evolution and cognition. Fuelled with a fascination of patterns of global emergent behaviour in systems composed of a large number of asynchronous interacting entities and inspired by various computational models of evolutionary processes, theory of hypercycles (Eigen and Schuster, 1979), symbiogenesis (Mereschkowsky, 1926) and autopoiesis (Maturana and Varela, 1980), this work tries to capture and unite various properties and intuitions associated with biological and artificial life through the EVMA. We have investigated areas of biologically inspired evolutionary computing models (Sipper and Tomassini, 1998), hypercyclic (Eigen and Schuster, 1979) and autopoietic modelling of the process of life (Maturana and Varela, 1980), the symbiogenesis theory of evolution (Margulis and Sagan, 1995) and theoretical aspects of computing, including the Turing model (Turing, 1936–7) and Gödel incompleteness theorem

---

1The Evolvable Virtual Machine concept will be abbreviated as EVM throughout the thesis.
2We will refer to such systems as massively parallel asynchronous systems.
The EVMA is a new framework which enables and facilitates research on massively parallel autonomic computing, autonomous systems and evolvable virtual machines.

In this thesis we try to communicate the experiences and knowledge gained from the study of massively parallel\(^3\) complex interactive systems in the context of computability and computational evolution. This work can be applied in the areas of computing and computational evolutionary systems. It can also be used in the area of massively parallel asynchronous systems in general, and as such, this work can be carried on, extended and refined in the context of theoretical computer science. Through a better understanding of biological evolution, new possibilities and opportunities are continuously arising within new multidisciplinary fields generally referred to as bio-inspired computational models. This study also provides novel modelling abstractions within the field of computational evolution. On the one hand, advances in theoretical evolutionary biology can improve and enhance computational evolutionary models. On the other hand, computational studies of artificial life and computational evolution may lead to better understanding of biological evolution itself and allow refinement of biological models. As in many other multi-disciplinary research programmes, various fields benefit from the refined models and from deeper understanding of certain regularities and principles that span multiple research areas. However, as each of the field uses its own vocabulary and formal semantics, discussing cross-disciplinary models presents additional challenges.

The habit of abstract pursuits makes learned men much inferior to the average in the power of visualization, and much more exclusively occupied with words in their thinking.

(Russell, 2003)

I have tried to avoid the pitfalls sketched above by Russell. This is mostly through providing multiple descriptions and analogies for a given concept in various systematic forms and with references to various fields, some of which might be well-understood by a reader.

These thoughts did not come in any verbal formulation. I rarely think in words at all. A thought comes, and I may try to express it in words afterwards.

(Einstein as quoted in Miller, 1984, p. 204)

\(^3\)A parallel system is a system that exhibits parallelism during the course of its operation, i.e. the computations of distinct elements progress at the same physical time. It is sometimes referred also as true-parallelism. It is the case with multi-core and/or multi-machine distributed systems. Concurrent processing refers to tasks that are carried on during the same time period, but not necessarily physically at the same time. Concurrency can be conducted on a uni-processor system through time-sharing of the CPU resources in such a way that multiple execution threads are executed concurrently (but not in parallel). The use of various forms of time-sharing, either through software or hardware Chip Multi-Threading (CMT) technologies relates directly to the concurrent processing paradigm. Multiple parallel systems that are located in physically distributed locations, are referred here as distributed systems. Distributed systems are generally a good example of a (true) asynchronous parallel system.
Although the formal language and the form of this thesis constrains the capabilities to express all the insights, thoughts and intuitions that I have developed during the course of my studies, I believe these restrictions to be related to the expressive capabilities of any formal system (such as traditional scientific methods and written language in general). Perhaps one day we will have more expressive ways of scientific discourse. Perhaps, too, we may be able to go beyond the Turing limit in our scientific methods, and the way these are communicated. For the time being, however, formal systems and Turing-like computation are our principal means of representing and expressing rigorous models of computation. Grand (2000, p.9) said: “The problem with insights is that they are like skills: you can’t use language to transfer them from person to person”.

1.1 Motivation

The main focus of this study is in the area of computational architectures suitable for modelling massively parallel systems capable of reflection and reification of its own structure. The secondary aspect that connects this research with various other disciplines is the applicability of such an architecture to the area of artificial evolutionary systems. The EVM architecture (contextualised in the Foreword section earlier) has implications in theoretical biology, computer science, and, to a certain degree, in philosophy. Below we will list the main motivations and briefly discuss them. Later, the main objectives will be presented in detail in the following subsections.

**Motivation 1:** *What can be computed by currently available physical computing machines, and by virtual machines? Is it possible to build computers capable of computing more?*

Can a computation performed by a collection of asynchronously communicating computational units be represented by a sequential Turing machine? If not, can the inherent properties of such a computation be utilised for any purposefully designed tasks or problem solving? How well-suited is the theory of computation based on Turing machines to concurrent computations conducted by networks of computing devices? Is it possible for a given computing environment to continuously redefine itself in such a way as to provide new computational capabilities? Is it possible to compute the “uncomputable”? How valid or plausible are the possibilities of hypercomputing: computing beyond the limits of Turing machines (Turing limit).

**Motivation 2:** *What is the relation between a theory of life and a theory of computation? Is it possible to make software systems truly autonomous, adaptable, and alive?*

---

4 See Chapter 4 for more detailed discussion related to the work of Turing and formal limits on computability of Turing machines. *Turing limit* relates to the computations that can in principle be carried out by a Turing machine. Any computation that a Turing machine is unable to carry out, is beyond a Turing limit.

5 Some philosophers and modern thinkers have proposed the use of art as a way for communicating ideas and scientific insights, possibly outperforming the ability of formal systems in this task (Witkiewicz, 1996; Bohm, 1996).

6 To calculate, guess, generate or use something that is not attainable through an execution of a single Turing machine.
How can we build software systems that closely mimic living processes, for example, software that behaves as if it is alive or software systems that adapt and evolve autonomously? How can we benefit from the recent advances in the life sciences in order to build bigger, more complex and more flexible software systems? Can we construct systems that exhibit self-reflective autonomy and evolvability? Is it possible to build a truly autonomous software system, yet maintain the necessary level of control that makes a given software system useful? What language is best suited for modelling, understanding and discussing the evolution of complexity in living systems? What can be used as a complexity measure in artificial and natural systems?

**Motivation 3:** What is the essence of life? What features distinguish animate from inanimate? How can we model life? How can we best model evolution and adaptation? Are the existing models sufficient, and if not, how they could be extended?

From the beginning of human history people have asked the question, “what is life?” This question has motivated and inspired many researchers from many disciplines. It has been asked by ancient Greek philosophers (Socrates, Plato), by physicists (Schrödinger, 1945), by biologists (Margulis and Sagan, 1995), and in recent years by computer scientists (Chaitin, 1979).

In this work we investigate the state of the art in evolutionary computational models. We propose a new model of computation together with a new methodology to discuss complex computational phenomena. Our new model is represented in the form of the EVMA. The prototype, a concrete Evolvable Virtual Machine Instantiation (EVMI) of some of the core features of the EVMA has been implemented and a series of experiments has been conducted. We have investigated certain properties of the proposed computing model and tested its performance on a representative set of problems. The main objective has been to investigate possible architectures that are capable of computationally expressing complex biological-life-like phenomena.

In this work we also look into various extensions to the traditional algorithmic theory of computation. This is done with evolutionary computation in mind and with the aim of better understanding the underlying mechanisms of biological evolution. In connection with this we analyse and reflect on various models of hypercomputation (see Chapter 4), and their potential relevance to the computational theory of evolution and life.

The three motivations are tightly interlinked, e.g., the second motivation cannot be pursued without a proper handling of the first one. Yet, the results and expertise gained in addressing the second motivation provide feedback to implement, modify and reify the models and methodology for the first motivation. This iterative process from the beginning influenced the progress of the research. The investigations following from the third motivation, of designing massively parallel asynchronous architectures, came as a result of earlier studies, which were influenced by the contemporary mod-
els of computation and recent discussions of alternative models of computation (Schrödinger, 1945; de Faria, 1988; Akl, 2005).

1.2 Modelling, scope and objectives

This work presents a novel way of representing and analysing a class of distributed self-organising, self-learning and adaptive phenomena through a specially designed computational EVMA. Our model is derived from the traditional models of stack-based virtual machines augmented with evolutionary computation and Tierra-like (Ray, 1991a, 1994) artificial life features. The work provides the means to discuss, analyse, experiment with self-reflective adaptable evolutionary processes in a massively parallel asynchronous computational system. We have analysed and investigated a class of computational models designed for studies of evolutionary computation. We have also investigated the possibility and utilisation of hypercomputing in the process of evolutionary computation. There are also open questions that we have identified and analysed through this research, such as the relationship between hypercycles and the autopoietic theory of life\footnote{The concepts of autopoesis and hypercycles are discussed in detail in Chapter 7.}, the role of meta-learning in the process of evolution and adaptation, and the relevance of hypercomputation to the computational modelling of various physical and biological phenomena.

The term hypercomputation is used to mean any alternative models of computations that go beyond or augment the sequential algorithmic notions based on a Turing machine. Reflection plays a central role in our computational evolutionary model, therefore, we put an emphasis on computational reflection mechanisms. By reflection we mean:

- introspection – the ability to inspect itself, i.e. to inspect the abstract model of a system from within the system itself;

- reification – the mechanism to make the abstract model of a system available in a verbose form for manipulation, and the ability to manipulate that model from within the system itself.

Traditional computational evolutionary systems, loosely based on general ideas of natural selection and random mutations, are limited in the way the detailed dynamics and relationships between interacting entities can be modelled and represented. The traditional evolutionary computation techniques produce a single solution for a pre-designed fitness landscape (Fogel, 1998). We are interested in a more detailed modelling of evolutionary processes, where the dynamics of multiple levels of abstraction (see Section 2.5.3) and between various interacting components are modelled explicitly instead of being collapsed to probability distributions. In particular, we seek a modelling paradigm in which the evolutionary system produces independent, asynchronously interacting and collaborating entities, whose interactions and dynamics can be observed and analysed. The presentation of the abstract EVM architecture and the concrete instantiation are provided in Chapters 2 and 3, respectively.
Note, that the term *machine* is sometimes contrasted with the term *living system*, as if these terms belong to two disjoint binary classes (Maturana and Varela, 1980). In our work this distinction is blurred. We claim that living systems at a certain level of abstraction may be seen as simple machines of a given complexity. We assume that certain organisations of machines can be seen as living systems. Hence, the *machine* in the name: Evolvable Virtual Machines. The machine metaphor used throughout this work goes beyond the traditional interpretations of individual mechanical machines, because a collection of distributed, asynchronously working machines goes beyond a uniform model of a single machine. Therefore, when using the term *machine*, we sometimes mean a single traditional mechanical machine with all the mechanistic properties traditionally ascribed to a machine. However, we also use the term *machine* for complex asynchronously interacting sets of machines, with certain properties that go beyond the traditional notions used to characterise a simple machine.

A Turing machine is an example of single machine in the traditional sense. Our general use of the term goes beyond the notion of Turing-like machines. In principle, to know the output of a collection of interacting machines one would need to know all the inputs, the state and the time delays between all the machines comprising the composite machine. For complex machine assemblies this task becomes intractable. Therefore probability distributions and pseudo-random processes are commonly used as approximations for certain tasks. In principle however, the system could be reified further and further and approximated to the desired level of accuracy with all the levels representing proper deterministic computational machines (see Section 1.3).

### 1.2.1 Technological advances

The rapid growth of complexity in different areas of technology stimulates research in the fields of computer architectures, adaptive computation and adaptive engineering. The mantra *one size fits all* is not adequate and not appropriate in virtually any aspect of the current IT landscape. General hardware designs are being replaced by more specialised solutions with customisable hardware and software components. The requirements for time-to-delivery in IT systems have been substantially shortened by contemporary market pressures. Technologies with increased flexibility and reduced time-to-market advantages are being preferred. There is a trend to use adaptable solutions in software and in hardware (e.g. through re-programmable solutions, such as Field Programmable Gate Arrays (FPGAs) and Transport Triggered Architectures (TTAs) (Corporaal, 1997)). A greater proportion of the tasks and deployments that have been traditionally hardware-based are now being achieved in software. This provides opportunities to software engineers and at the same time also presents multiple challenges. The engineering capabilities of fitting more transistors on a single chip present multiple opportunities to facilitate computational architectures that differ substantially from traditional uni-processor systems. The hardware manufacturers and market pressures fundamentally change

---

8Due to the asynchronous character of the interactions. See Chapter 4 for detailed exploration of this topic.

9Not governed by a single clock with an enumerable set of states.

10Note, that this implies that each reification process leads to a creation of a new machine level.
the computing architectures. It is often no longer sufficient to deploy an old software system on a new hardware platform. For efficiency reasons and performance considerations, applications must be either rewritten by programmers, or adapt themselves to take full advantage of the hardware capabilities. In particular, operating systems and applications must take advantage of the increasing amount of hardware parallelism that is available in order to fully benefit from the technological progression. In addition, there is an observable transition in perspective from unified, monolithic software models, towards loosely coupled, concurrent, autonomous and asynchronous computing architectures. This transition will become even more evident when the multi-core and multi-processor architectures become widespread in consumers’ personal computers, workstations and embedded computing devices. As a result, computer architectures and software systems are rapidly moving from single-CPU, monolithic software models towards multi-core, multi-threaded, asynchronous, ad hoc and massively concurrent computing. We need to design tools, methodologies and architectures to help us manage the increasing complexity, and we need to design methodologies and control the behaviour of completely new massively parallel asynchronous programming paradigms.

To tackle these challenges, automated tools and more adaptive and autonomous software/hardware systems are required. Adaptive software models refer to generic concepts such as autonomy, adaptability, and evolution. Although a vague intuitive understanding of these concepts is shared among researchers from different fields, these concepts are not yet fully understood in all their consequential detail, especially in the context of computational systems. In this work we have addressed some of the questions related to open-ended evolutionary processes (Standish, 2001; Maley, 1999): essential properties, minimal requirements, architectures, models and the notions of evolvability itself. There are many disciplines concerned with studying various aspects of adaptability and autonomy – the work in these areas will lead to:

- better understanding of the scope and limitations of autonomous and adaptive software systems;
- progress in automatic evolutionary designs of computational machines;
- better insights into computation and better understanding of hypercomputation;
- better suited architectures for massively parallel asymmetric and asynchronous software systems.

1.2.2 Meta-evolution

*Evolutionary Computation* (EC) is an umbrella term that covers multiple methods involving combinatorial optimisation problems. The majority of EC techniques use information-centric methods that mirror the process of the Darwinian (or to be precise, neo-Darwinian, see Chapter 5 for a detailed discussion) theory of random mutations and natural selection. This is visible in well-established computational models of evolutionary processes, such as Genetic Algorithms (GA) (Vose, 1999), Genetic
Programming (GP) (Koza, 1992), Evolutionary Computation (Fogel, 1998), and their variations, such as assorted Artificial Life systems, for example Avida (Adami, 1999) and Tierra (Ray, 1991a). These systems produce a single monolithic solution for a single pre-defined fitness landscape (for detailed explanation of concepts related to evolutionary computation, such as fitness landscapes, genotype, phenotype, mutation and selection, refer to Chapters 5 and 6). These systems also operate on a flat fixed mapping between genotype (a specific allelic makeup of an individual) and phenotype (any detectable characteristic of an organism determined by an interaction with an environment). Simple single-layer evolutionary systems based on random mutation and selection have been shown to be insufficient (in principle) to produce an open-ended uniform evolutionary process with potential multiple levels of translation (Wright, 1931; Eigen and Schuster, 1979).

Our work proposes an alternative path based on our own EVMA-based model of computation and evolution. The intrinsic properties of hypercycles (Eigen and Schuster, 1979) allow us to model and analyse structures that evolve into higher levels of complexity, analogous to multi-level, or hierarchical evolutionary processes. Combined with principles of autopoiesis (Maturana and Varela, 1980) and symbiogenesis (Mereschkowsky, 1926), our focus is on developing a methodology and framework to model and experiment with open-ended hierarchical multi-level structures of self-maintaining ensembles. In other words, we investigate the process of evolving such hierarchies and societies of interactive, asynchronously communicating computational units. One of the principal tasks of the present work is to develop an architecture that facilitates experimentation with adaptable and evolvable software systems on multiple levels of abstraction. The goal is to provide a framework for the development of techniques that automate the creation and control of software systems (and computational systems in general) composed of many independent autonomous processing units. This is to help in the utilisation of new, highly parallel multi-core computational architectures. It is also to help in building software systems with complexity levels exceeding those currently available. The EVMA model facilitates experiments on software and hardware architectures suitable for such complex systems – those oriented towards massively parallel architectures and more efficient, flexible, adaptable and evolvable virtual machines.

1.2.3 Theory of computation

Distributed and interacting systems. The existing theories of mechanical computation were established in the 1930s resulting in several equivalent models of algorithmic computation. There are three well-known models: Turing machines (Turing, 1936–7), recursive functions (Kleene, 1936) and the lambda calculus (Church, 1941). All of the various models of mechanical (algorithmic) computation have been demonstrated to be equivalent to each other (Church, 1941), being effectively equivalent

---

11 An allele is a viable coding that occupies a given position on a chromosome of an individual organism.
12 According to the autopoietic notion of living systems (Maturana and Varela, 1980), phenotype is epigenetically being constructed throughout ontogeny. The distinctions that define any particular trait at any particular state of the organism’s life cycle are constructed from outside of the organisms itself, by the external observers.
to a Turing machine (Gandy, 1988). However, there are some tasks that cannot be computed on a Turing machine. One such task, known as a “halting problem”, is the fact that it is unknown in principle if an arbitrary Turing machine will stop its execution or execute instructions indefinitely. One can say, therefore, that a Turing machine has a particular computational power and cannot compute certain classes of tasks that would require larger computational power (Turing, 1936–7; Copeland, 2004). For those larger tasks, different kinds of machines would need to be used. All mechanical computing devices conform to the Turing machine model, and there currently are no known theoretical frameworks that would enable us to exceed the known computational limit of existing physical and mechanical devices (Church, 1941). However, theoretical models have been proposed and alternative models of computation are emerging (Copeland, 1997; Calude, Casti, and Dinneen, 1998; Wegner and Goldin, 2003; Teuscher, 2006). These models provide new foundations and help us understand and/or expand computational architectures in use today. The research on theoretical aspects of hypercomputation will give us insights and it will provide progress in the theory of computation. It will also help in the design and implementation of alternative computational architectures, suitable for highly concurrent distributed systems.

**Hypercomputation.** Current research in theoretical biology and evolutionary computation assumes (most of the time implicitly) that the natural processes of evolution may, with arbitrary precision, be modelled by a process executed on a single Turing machine or computational device of equivalent computational power. This however, is a strong assumption that presents a major theoretical and practical challenge to the validity of the field of computational biology and the evolutionary computation as a whole.

In this work we investigate some of the results in the field of hypercomputation and their implications and effects in artificial life and evolutionary computation. We propose an extension to evolutionary computation theory and introduce some linkages with the theory of hypercomputation. These two areas: modelling complex systems and hypercomputation are, in our opinion, interlinked. Better understanding of hypercomputation may lead to progress in resolving the fundamental issues in modelling adaptive and evolutionary systems. Because artificial life and computational evolution presuppose particular computational architectures we argue that any theoretical advances and discoveries in the field of hypercomputation should be taken into consideration in the fields of artificial life and computational evolution. In our current work, we investigate a computational model based on a collection of parallel asynchronously operating machines, each of which is capable of exhibiting Universal Turing machine-like processing. Due to the asynchronous and interactive nature of the model, it is plausible that an appropriate implementation could exhibit computational capabilities that would be difficult or impossible to be replicated on a sequential Turing machine. These issues and properties are partially motivating the current scope of this work, in the context of theoretical computability theory.

**Self-reference and meta-modelling.** One of the fundamental properties of the EVM architecture
is the lack of distinction between a base level and a meta level (see the discussion on meta-models and reification in Chapter 8). Traditionally, systems are modelled through a set of concepts that form a base-level - a set of first-class objects of the given model. The meta-model, which defines the semantics of the base-level object and their relationships is expressed on another level, the meta-level. Usually, one clearly distinguishes between the base level and single or multiple meta levels, with clearly defined boundaries between the levels, because the distinction can make the theoretical analysis simpler, tractable, and avoids some properties that are difficult to be modelled and managed by a formal theory. As argued by some contemporary authors e.g., (Nagarjuna, 1994), the modularisation taken as a strict encapsulation of information is not a desirable feature when certain classes of complex information processing tasks are concerned. Therefore, in the EVM model the strict modularisation of levels has been relaxed, and the meta and base levels interweave and complement each other, without clearly defined boundaries. This allows easier information flow and feedback-loop formation processes, which would not be possible otherwise. It is our objective to investigate the relevant features and implications of such a computational architecture with the emphasis on its applicability to the fields of artificial life and evolutionary computation. The implementation issues and their applicability to a general purpose computing architecture are also of significant importance and are discussed in this thesis.

1.2.4 Philosophical objectives

The nature of the process of life, process of evolution and modelling the Universe as a whole has a fundamental place in philosophy. Recently, these questions have been of interest not only to philosophers and biologists, but also to computer scientists. New multi-disciplinary research themes are emerging and efforts are being made to bridge various research areas that investigate the process of life and evolution. The common theme bridging the various disciplines appears to be the theory of computation. For example, through Digital Philosophy and Digital Physics (Zuse, 1970a; Fredkin, 1992) various areas of physics and computer science have been bridged. The main goal in these fields is to provide minimal models capable of expressing and simulating given aspects of physical reality. By pursuing research in the area of virtual machine architectures, we also aim at building abstractions and theoretical foundations for understanding properties of evolutionary processes and the process of life. One of the main features of the EVM architecture is its self-reflective nature. We present an innovative model that goes beyond current cellular based systems and captures the hypercyclic and autopoietic nature of reflective hierarchical computational systems. We believe that further research based on the presented model will provide a better understanding of a certain class of emerging phenomena. In particular, we address phenomena that are characteristic of life processes and biological evolution.

There are associations between the EVMA and the philosophical movements of post-modernism and hermeneutic phenomenology. The main representatives of the movement of hermeneutic phe-
nomenology are Heidegger (1959), Gadamer (1994, 1997), and Ricoeur (1981). The hermeneutic movement postulates that human beings are in a continuous process of interpreting the world and themselves – this defines the fundamental structure of existence. Those notions are extended further by Gadamer, who formulated that understanding is an event or experience that we undergo. The central concept is the interpretative character of knowledge. This orientation is especially evident in the work of Heidegger, who argues that all description is always prepared as interpretations. Every form of awareness is interpretative. In the EVMA all the interactions between processing elements must be interpreted in the context of the lower level virtual machine on which a given level is being executed. In turn, the lower level itself has to be interpreted on another level, and so on. The sequence of symbols must always be interpreted in the context of a certain virtual machine. The process of interpretation may (but does not need to) modify the actual interpretation context, leading to a different interpretation of the same sequence of symbols in the future. This allows the hermeneutic circle (Heidegger, 1959) to be an important element, strongly influencing the EVMA. The concept of hermeneutic circle concept is based on the notion that one can only know what one is prepared to know, in the terms that one is capable of understanding. In phenomenological hermeneutic theory the hermeneutic circle does not close off on itself like a “vicious” circle. Instead, it opens up and becomes more like a spiral, due to the symbolic and self-reflective nature of the being. Similarly, in the EVMA, the interpretative power of a given level is constrained by the virtual machine on which that level is being executed, and in which context a given sequence of symbols is being interpreted. However, owing to self-reflective capabilities, a given level can modify (expand/reduce) the execution level and manipulate the context, providing alternative, expanded or reduced interpretations for the symbols that are being interpreted on that particular level. This, in turn, presents interesting questions regarding the notion of self and self-identity. The EVM model instead of having a unified identity and centered self, presents a sense of fragmentation and de-centered self, with multiple, sometimes even conflicting identities. Those links of the EVMA to the roots of hermeneutics allow the overall EVMA to be discussed in terms that are characteristic of modern philosophy. Some of these issues are discussed in the Chapter 2 and later in Appendices B and C.

1.3 The thesis

Our proposed abstract architecture, referred to generally as the EVMA, (Chapter 2), offers a model of living systems as an interconnected web of networks of encapsulated processing machines. The processing machines have a specific organisation that supports hierarchically organised computation on multiple abstraction levels carried out in parallel. The model incorporates notions of parallelism and on reflective capabilities of processing machines. The EVM universe forms a multitude of autopoietic hypercycles, each one with the characteristic autopoietic and symbiotic patterns of organisation. The symbiotic relationships and other biologically inspired mechanisms such as symbiogenesis are introduced in Chapter 5. All the interacting machines maintain themselves in the state of self-organised
criticality (SOC) (Haken, 1983; Bak, 1996), where all the possible evolutionary trajectories (all con-
current evolutionary choices) are being represented (or approximated) by the appropriate computa-
tions. These computations operate on different levels of abstraction, from machine level, through
the meta-level, to \( \text{meta}_0, \text{meta}_1, \ldots, \text{meta}_n \)-levels (see Chapter 8). The EVMA theory provides a
meta-evolutionary framework that is capable of reflecting on its own organisation (see Chapter 6 and
7). The theory represents a holistic and deterministic view over different evolutionary processes. In
principle, the evolution in the EVM universe does not require the use of any intrinsic probabilistic
models. There are no probabilistic constructs similar to other forms of any evolutionary models (such
as random mutations). Instead, EVMA provides the framework and explains the methods to reify all
the choices on a particular level, with concrete sequential and deterministic programs. In that sense,
the EVMA can be summarised as a non-probabilistic model of evolutionary computation. Further
studies are needed to establish the relationship of the model with biological evolution. Due to phys-
ical limitations of participating (finite) computational entities (cells) the deterministic model must,
when implemented presently, be locally approximated by certain probabilistic mechanisms. This is
an emerging property and not an assumed \textit{a priori} feature of the model. The local probability dis-
tributions also free up an artificial complex system modeller to include more and more reified lower
levels of the represented phenomena within the model itself. The model allows a dynamic collapse
of a given layer to a probability distribution where detailed computation for a given computational
layer is too long or too costly in terms of necessary resources when compared to an abstractly defined
benefit of such a computation. In other words, unlike neo-Darwinism, the proposed model does not
enforce any probability distribution on any particular level. In that sense the EVM model can be seen
as a meta-evolutionary theory.

1.4 Current research within life sciences

The research programme of modelling life, i.e. all the efforts aimed at explaining the phenomenon of
biological life, is split into three categories. These categories are somewhat independent, yet over-
lapping in various aspects. In one of the categories, the main research objectives are focused on the
physical and chemical properties of living systems. Researchers are looking for substances and ma-
terials that distinguish living from non-living matter. This direction is pursued mainly in research
fields such as biology, organic chemistry and classical physics (Randerson, 2007). The second cat-
egory consists of researchers not focused on matter as such, but on the organisation and organising
principles that distinguish life. In this category, researchers argue that the substances that make up
living systems are not important in themselves. This group of researchers maintains that to make
progress and to understand \textit{life}, one has to investigate the forms and patterns that could be classified
as characteristics of living systems. This line of work is pursued by cybernetics, theoretical biology,
and certain fields from mathematics and information sciences. The third category is a specific subset
of the second category – it focuses around modelling the processes with the use of Turing computa-
tional models. Researchers from this category are interested in self-organising distributed systems, self-assembly, models of evolutionary computation (EC) and artificial life (ALife). These researchers are trying to model adaptation and evolution through synthesis via different computer models and computer simulations. The fact that ALife research is pursued mostly through experimentation and simulations is somewhat controversial. Some claim that proper mathematical models are the only rigorous way of progressing with theories and knowledge, whereas others claim that due to high levels of complexity and intractability of any mathematical models of living systems, only experimentation and simulation-based studies are possible. Our work fits into this third category, although strong linkages with the second category are also established.

Most of the existing models in the third category derive from the original Darwinian and neo-Darwinian notion of evolutionary processes. Our work expands on those models. This is achieved through our study of natural evolutionary processes and advances in theoretical evolutionary biology. We expand the existing model of artificial life through the notion of massively parallel processing, hypercycles and autopoiesis and hierarchically organised virtual machines. These concepts will be discussed in detail in Chapter 7. These are unique elements and features that distinguish our architecture from the other existing models.

There is ongoing work carried on in all three research areas. The current models, however, fall short in capturing and explaining the process of life in sufficient detail so that an integrating theory spanning the disciplines would be possible. Current theories in evolutionary biology, artificial life and computational sciences do not match well together, even though substantial overlaps exist. Different research fields continually need to modify existing models and understanding, based on advances in other fields. Recently, thanks to advances in physics, mathematics and computer science, we have better insights into some of the underlying processes and patterns of organisation within living, adaptive and evolutionary systems. In physics, the insights derive from the research on complex dynamical systems research; in mathematics from chaos theory, and in computer science from studies in the fields of artificial life, evolutionary computation and from uncomputability vs. computational universality (Wolfram, 1984; Krauss and Starkman, 2005).

The existing artificial evolutionary models (Fogel, 1998; Koza, 1992) are characterised by a fixed objective function and/or fitness landscapes. This is done to simplify and manage the complexities of the evolutionary processes. Certain essential aspects are purposefully not being taken into account and not captured in the artificial systems. There is an ongoing search for a unifying element of the theory, that spans various disciplines. Also, not all of the assumptions regarding the modelling of living systems may be justified. We claim that the current Darwinian formulation is a good approximation of some natural phenomena. However, other approximations are possible, and might be necessary. We propose a different set of assumptions, that are, in our view, more accurate. We formulated our model for a discrete computational universe. In simple terms, the basic ideas behind our formulation are as follows:
1. there is no need to postulate inherent randomness to explain the dynamics and behaviour of an evolutionary system; instead of inherent randomness, we can try to model the complexity of the system to a desired depth; we need to use randomness only to provide input where modelling of the features stops to be relevant;

2. all the emerging probability distributions and search biases (see Chapter 8) are driven by the nature of the processes involved;

3. there are some uncomputable elements on different levels of any evolutionary process that are not random in nature, but can only be approximated by a random search (due to their uncomputability).

Our formulation assumes that given a complete knowledge of all the components of a given deterministic\textsuperscript{13} system, in certain circumstances, one will be unable to predict\textsuperscript{14} the future state of this system. In computer science this result has been established through the notion of a halting problem on a Turing machine. The more general form is presented through the Gödel Incompleteness Theorem (Gödel, 1934) which applies to any formal axiomatic system. In short, the halting problem demonstrates that there are computer programs whose outcome cannot be calculated or predicted without actually executing the program itself. This is a crucial and important property of computational systems. It influences other fields as well, and it puts a question mark on scientific reductionism in its most orthodox form. Our view differs from the Cartesian and Newtonian world view. Traditionally, it had been thought, that given a deterministic system, with all the rules that govern the low-level interactions and the initial parameters, the future state of the overall system could be calculated, at least in principle. However, even in computable universes\textsuperscript{15}, there might exist non-computable dependencies\textsuperscript{16}, and the exact predictability of future state in a deterministic computational system is impossible (see Gödel Incompleteness Theorem). We assume for our model that the real world is computable in a formal mathematical sense. However, due to Gödel’s result we know that uncomputable relationships are abundant in such a model (and in nature, too). We postulate that living systems are adapted to deal with and to approximate these uncomputabilities through various learning techniques and adaptable asynchronous computational models. We have explored and investigated in our studies on evolvable virtual machines some of these, including trial-and-error machines, simple brute force, random search\textsuperscript{17}, and meta-learning. Our assumptions about the reality and the nature of the universe have many philosophical implications. However, our primary focus of this work is the

\textsuperscript{13}By deterministic we mean a system in which all the actions are strictly guided by the rules of the system itself, and no randomness is involved.

\textsuperscript{14}By predict we mean here calculate ahead of time, calculate the future state of the system before the system reaches that state itself.

\textsuperscript{15}See Section 4.4 for detail.

\textsuperscript{16}E.g. tautologies that cannot be deduced from the axioms, but can be observed to hold through evaluations and sampling.

\textsuperscript{17}Randomness is used as an approximation of computable (though not specified in every detail) or uncomputable phenomena.
study of a class of computational systems related to meta-learning and asynchronous massively parallel computational models. The philosophical implications are not described here in detail. Interested readers can find some further discussion on the topic later in the thesis. Exploration of this topic of computation is presented in Chapter 4 and more philosophical implications in Appendix C.

Given the current state of the art, we propose a new architecture to investigate and talk about complex computational and evolutionary systems. The results of our investigations provide an extension to the basic traditional notions of organic and artificial evolutionary models. It is our belief, confirmed by the experimental results, that our EVMA is a good way of modelling and investigating evolution and life. This is via a system of independent processing units, all capable of universal computation (in a Universal Turing machine sense, see Chapter 4 for detailed discussion), interacting asynchronously and collaborating together in a massively parallel and interconnected system. These components can form emerging circles of interdependencies to maintain and sustain themselves – this is achieved through hypercycles (Eigen and Schuster, 1979). At different levels of abstraction, the individual components need to have sufficient capabilities to reify and reflect on their own computations.

1.5 Organisation of this work

The EVMA is discussed in this thesis on multiple levels and from different perspectives. Because these perspectives often overlap each other, it is difficult to partition the discussion clearly into individual and separate parts. The detailed presentation of the architecture (Chapter 2) and its concrete instantiation (Chapter 3) is not possible without a discussion on autopoiesis and hypercycles (Chapter 7), which in turn require the detailed understanding of theories of biological evolution (Chapter 5). The computational perspective discussed in Chapter 4 overlaps with some of the philosophical discussions. The artificial evolution chapter (Chapter 6) references many biological concepts from natural evolution (Chapter 5). All these concepts and relationships form a highly non-linear structure, that is difficult to be placed into a linear structure in a written document.

The main chapters discuss different subjects related to the main thesis of this work. These should be read in sequence provided in this thesis. However, some of the main chapters are self-contained and could be read individually in any order (Chapters 4, 5, 6). The chapters provided in the Appendix are extensions of some of the topics discussed in the main part of this work, and may be read individually in any order.

In Chapter 2 we present the main abstract architecture, and in Chapter 3 we discuss the EVMI model for a single cell, together with the discussion of the EVMI assembly language.

In Chapter 4 we provide an introduction and elaborate on the modern theory of computability. We introduce some of the essential concepts, provide a brief history of the field, and discuss the main

18For a detailed critique of some of the existing theories of life and evolution, see Chapter 5, which reviews the state of the art in models of life and evolution.
contemporary notions of computation and computing machines. We also discuss some aspects of recent theories, and sketch directions of further research.

In Chapters 5 and 6 we introduce the reader into modern theories of evolution, and provide different aspects of biological evolutionary theory in appropriate historical contexts. Then, we provide theoretical arguments and experimental evidence for some of the shortcomings and insufficiencies of existing computational evolutionary models. In particular, we explore the evolutionary computation models derived from neo-Darwinism and discuss the problems and limitations of these models. The analysis and critique of the existing artificial evolutionary systems motivates the main efforts of this work.

In Chapter 7 we present a unified theory of autopoietic hypercycles. Chapter 8 consists of a discussion on machine learning, multi-task and meta learning. In Chapter 9 we discuss different experiments and experimental setups that we have used to evaluate the current EVMA. Some of the experiments have been used to reify the EVMA, and some were used to investigate different properties of the EVM model.

The entire work is summarised in Chapter 10. This chapter provides the summary of the results and contains a discussion on different experimental issues. It also draws future research avenues for the EVM-based evolutionary studies.

1.6 Summary of the contribution

The research work presented in this thesis has four main contributions. It provides:

- an overview of multiple computational models in the context of biological life and evolution;
- a new biologically inspired, massively parallel asynchronous computing model that follows Forth Virtual Machine traditions;
- a concrete implementation of the model;
- experimental results based on the current implementation.

The EVMA exhibits properties characteristic of complex living systems, i.e. self-maintaining adaptable and evolvable systems. The EVMI provides a framework, methodology and language for more refined investigations into living, evolutionary and adaptive systems. It also provides a biologically inspired model of evolutionary computation that departs from fixed encoding and fixed fitness function models, providing a tool for explorations of open-ended evolutionary systems.

This work provides a novel synthesis of several independently developed theories, most notably that of hypercycles (Eigen and Schuster, 1979), autopoiesis (Maturana and Varela, 1980), and symbiogenesis (Mereschkowsky, 1905). When adapted in unison, they augment the current neo-Darwinian
theory of evolution (Mayr and Provine, 1980) and use it as a model of adaptable evolvable software architecture in a meta-evolutionary setting.

The main contribution is a new massively parallel asynchronous computational model, based on the theory of autocatalytic hypercycles and symbiogenesis, and loosely inspired by the stack-based (Forth-like) Virtual Machine model. We propose and investigate the EVM model as a suitable and appropriate architecture for understanding, modeling and discussing the evolution of complex systems. We believe that the EVMA is an adequate model for a class of complex phenomena in massively parallel asynchronous and asymmetric software systems. We assume that life itself is a continuous property that can be exhibited by a collection of inanimate components composed and organised in a particular way. We claim, that this particular organisation, characteristic of living organisms, can be observed on different levels of abstraction. We put emphasis not on individuals, but on a hierarchy of collections of individuals (on societies and collaboration). The traditional perspective on life and evolution based on coarse-grained monolithic single-layered individual organisms (represented through the phenotype abstraction), is replaced by a highly asynchronous and distributed collection of interacting hierarchies of entities (like individual organelles, cells, organs, bacteria). We use a computational hierarchy and societies of computational units, that together create something that can be perceived as being alive, i.e. exhibiting certain properties characteristic of the process of life. We propose a new model of computation, the EVM universe that models these complex interactions. The central element of this model is its distributed and asynchronous character, interwoven with reflective capabilities, adaptability and evolvability.

Apart from general theoretical and technological contributions, this work represents a new perspective on the notion of living organisms, evolution, and the concept of self. This perspective has consequences especially in biology and in philosophy. Unlike traditional theories that model self as a self-contained unity, we believe it is more accurate to treat self as an emergent property of a society of elements, which only appear as a cohesive and unified entity to the observer, and only in certain circumstances. Our new model, when applied to software systems, provides a platform for experimentation with self-maintaining autonomous software systems and automatic life-long reinforcement learning and adaptability.

We have conducted a review of contemporary theoretical models of biological evolution, and we have investigated the possibilities of an integrating meta-evolutionary computational architecture and subsequently we have designed and implemented a new model based on the EVMA. In so doing we have designed a novel theoretical architecture for representing a class of complex dynamical phenomena. We used the notion of massively asynchronous interacting distributed computation, with a special hierarchically organised virtual machine (VM) to conduct experiments on our new meta-evolutionary model. Our computing architecture provides a novel way of representing massively parallel asynchronous and hierarchically-organised computations. The developed virtual machine, together with a sequential program that represents the state of the VM and programs executed hereon,
facilitates compact and expressive notation. This notation can be used to provide a flexible, robust and adaptable computing framework for use in the field of artificial life and evolutionary computation. Our computational architecture also extends the algorithmic notion of computation\textsuperscript{19} and concentrates on models based on asynchronous trial-and-error machines\textsuperscript{20}.

We discuss the possibilities of hypercomputation and its relation to the fields of artificial life and computational evolution. We provide a framework in which such experimentation could be conducted, and we propose new directions for further research in this area. There are many aspects of computation and hypercomputation in biology and philosophy that are not discussed in this work, and that remain to be explored in more detail elsewhere.

In developing the theoretical EVMA, we have gone through several iterations and tried out several different implementations. During the course of this study we have investigated some existing architectures in evolutionary computation and self-adaptive self-learning software systems. In particular, the current version is based on the integration of ideas from different fields and research projects in which we were involved. Most notably, we have been inspired by recent advances in molecular and evolutionary biology and alternative evolutionary computing architectures.

In the context of computational theory, we have integrated several research areas: artificial evolutionary systems, theoretical biology, hypercycles and autopoiesis. The model presented in this work can be extended and applied in various forms to different areas of machine learning, evolutionary algorithms, computer design and computing architectures. The EVMA is, potentially, a universal framework for experimentation and provides a language for expressing different types and aspects of complex self-organising, self-adaptable and evolutionary systems.

The experimental results, conclusions, and intuitions have philosophical implications related to the concept of concurrency, modelling abstractions, and fundamentals about physical reality, concepts of countable and uncountable infinities and infinitesimals, and others. These are only briefly discussed in this work. Some areas that need further thorough analysis are the linkages of the EVMA with hermeneutics (Ricoeur, 1981) and with David Bohm’s interpretations of the quantum theory (Bohm, 1957, 1995). These are left for further studies.

\textsuperscript{19} Algorithmic computation means that a given task can be effectively solved given a finite mechanical procedure, that when followed step by step, leads to the solution.

\textsuperscript{20} A trial-and-error machine is a machine that follows an arbitrary random search and continuously proposes new hypotheses and refines them based on the feedback it receives from its environment. If the machine remembers the hypotheses that were accepted for a given state of the environment, it then can solve tasks to which no previously known solution existed, or to which no known computational solution exists. Note, the machine cannot be guaranteed to solve anything in advance. But, once a solution is found, the machine would simply reuse it, as if it was a computable function.
Chapter 2

EVM Abstract Architecture
The human mind has first to construct forms, independently, before we can find them in things.

Albert Einstein.

We’re all one thing, Lieutenant. That’s what I’ve come to realize. Like cells in a body. ’Cept we can’t see the body. The way fish can’t see the ocean. And so we envy each other. Hurt each other. Hate each other. How silly is that? A heart cell hating a lung cell.

Cassie, “THE THREE” screenplay, from “Adaptation” (Kaufman, 2002).

2.1 Introduction

This chapter contains the presentation of the Evolvable Virtual Machines Abstract architecture (EVMA). We provide a detailed discussion of the abstract model, its implications, generality, and adaptability. This chapter is followed by a description of several concrete prototypes, developed and tested during the course of the project. These are referred to as Evolvable Virtual Machine concrete instantiations (EVMI). The instruction set and one of the concrete implementations that we have worked with, will be discussed as the EVM Kernel (EVMK). The EVMK is the central part of a single processing cell, and we will use the term EVMK and cell interchangeably. Further description and analysis of the EVMK is provided in Chapter 9.

The goal of the EVMA is to provide vocabulary and modelling abstractions, tools and the constructs that can be used for prototyping concrete evolvable environments. The EVMIIs provide an experimentation platform for the generic meta-evolutionary and meta-learning model. The EVMK is also used as a bootstrapping implementation for development and experimentation with concrete instantiations. The ultimate goal is to create an environment in which the EVMI (i.e. the evolution and adaptation of a given EVM cell) construction can be automated sufficiently so that new virtual machines can be produced autonomously, at runtime, and spontaneously, by the working system.

We present the general design of our EVM architecture first. The requirements, general abstract model and compulsory and optional features are discussed. This is augmented with a discussion of the features that we have investigated to date and their possible extensions.

2.2 General ideas and inspiration

Computation, in general, is based on a collection of rules that manipulate symbols in a certain pre-defined way. Each one of the rules is fixed, but the potentially infinite arrangements of these rules create endless trajectories of what the actual computation performs. In general, different models of computation are capable of expressing the same set of trajectories. However, different models vary
in how long it takes to express a given trajectory. For some trajectories some models are shorter and
others are longer. The mathematical analysis of these aspects will assure us that the difference is
only in a multiplicative constant. However, for many practical purposes this multiplicative constant
appears to be prohibitively large, promoting some and not other models of computation. The property
of computable systems to differ only in a multiplicative constant is often referred as the universality
principle. All these properties and issues will be discussed in detail in Chapter 4. Readers unfamiliar
with the theory of computation should read Chapter 4 first.

Most models of computation, regardless of their actual detail, have one thing in common. They are
fixed, as if the rules of the game were written in stone and cannot be changed during the computation
itself. Imagine what would happen if the rules could be changed, adapted, modified, extended, or
trimmed. The investigation of such a computational model is at the heart of the EVMA. The main
research objectives for EVMA design are:

1. investigate the features needed for a virtual machine with the ability to self-reflect and re-
   implement itself;

2. explore the adaptability and how to evolve computational structures;

3. provide computational support for hierarchical decomposition and hierarchical abstractions
   within the model of computation itself.

We will now discuss the above objectives and the existing state-of-the-art, forming a source of
inspiration for this thesis. The first objective is to provide a robust and flexible virtual machine, which
can manipulate its own underlying implementation in order to adapt itself to the needs of a particular
running program. Existing computational architectures provide little support for such an adaptation
of the underlying virtual machine. We have investigated different aspects of existing virtual machines,
including, but not limited to, the design of a stack-based Forth virtual machine (Moore and Leach,
1970), Smalltalk (Goldberg, Kay, and Xerox Corporation, 1976), Java (Venners, 1999), and .Net with
the Common Language Infrastructure (Miller and Ragsdale, 2003). We have also looked into the
design of modern stack-based and register-based Central Processing Units (CPUs) and investigated
the possibility of designing self-adaptable hardware architectures.

The second objective is to investigate different properties that influence the evolvability and posi-
tible meta-evolutionary\(^1\) computational models. We are interested in properties that increase evolv-
ability of a particular genetic language. Some work in the area of artificial life and evolutionary
computation on evolvable languages has contributed substantially to the current EVM design. Most
notably, we used some of the ideas from the Tierra (Ray, 1991a,b) and Avida (Adami, 1999; Ofria and
Wilke, 2004) systems, the ADATE system (Olsson, 1995), Grammatical Evolution (Ryan, Collins,

\(^1\)By meta-evolution we mean the process of evolving evolvable structures, that is, structures that are easily evolvable
and adaptable. We will discuss these concepts in more detail in Chapter 6.
The third objective is related to the second. The earliest attempts at multi-level evolutionary computational designs were undertaken relatively recently (less than 15 years ago) and we were able to trace these ideas back to Juergen Schmidhuber’s diploma work. Schmidhuber himself reports (Schmidhuber, 1987):

Pages 7-13 of Schmidhuber’s (sic) diploma thesis (Schmidhuber, 1987) are devoted to a more ambitious self-improving GP [Genetic Programming] approach that recursively applies metalevel GP (first introduced here) to the task of finding better program-modifying programs on lower levels – the goal was to use GP for improving GP.

This line of research has also been taken up by Olsson’s ADATE system (Olsson, 1995). Meta-level evolutionary systems provide a sound and important contribution to the understanding and synthesis of artificial and biological evolutionary processes. Their theoretical frameworks are difficult to evaluate empirically, however, due to high processing requirements. It is important to note that only two-level (meta-level and base-level) approaches have ever been studied empirically in automatic program generation (among others, by Olsson (1995)). Only the work of Schmidhuber expands this notion into the meta-meta-…-level architectures. Theoretical work based on Levin’s search (Levin, 1973; Hutter, 2002; Schmidhuber, 2002a) is valuable and provides a background for further exploration. The Optimal Ordered Problem Solver implementation (Schmidhuber, 2002b) (OOPS) has been one of the inspirations for the EVM architecture implementation.

The original OOPS system has been criticised as not having much practical value due to the extremely high computational costs when applied to even “simple” computational problems. Such a critique, though valid for complex meta-programming paradigms in general, is missing an important point: for simple problems it does not make sense to use program generator techniques, or to use Levin’s search for an optimal algorithm. However, beyond a certain complexity threshold, the exploration of program spaces may be a better, more efficient way to find a solution to a given task. The main point of meta-level automated program generators is to investigate ways for more efficient and effective program generators. One of the meta-programming aspect is related to the modelling of tendencies and algorithm generators. The second aspect is to make sufficient effort, through implementation techniques and through incorporating expert knowledge, such that the final system is efficient and can be effectively employed to tackle real-life problems.

The EVMA departs from the pure deterministic search processes of existing meta-search systems, such as those based on Levin search (Levin, 1973). For example, the OOPS system (Schmidhuber, 2002b), conducts bias-optimal exhaustive search in the program space. Not only is the search exhaustive, but also all the partial results are assumed to be stored indefinitely in an unlimited storage. Those properties make OOPS non-applicable to most real-life computational problems. The EVMA on the other hand does not assume the following:
• deterministic exhaustive search

• infinite storage for partial solutions

This means, that a) EVMA employs stochastic methods to explore (only) part of the total search space. This may be taken arbitrarily close to exhaustive search by regulating the actual search space that a given method explores; b) EVMA needs to dispose of and disregard some of the previously constructed structures due to space limitations. As in case a), for many problems that can incorporate all the partial solutions within the storage, this can be treated as effectively unlimited storage capabilities. Based on these two main points, EVMA can be treated as a generalisation of generic bias-optimal search techniques, such as OOPS for example. In fact, EVMA can use various stochastic search techniques to explore an unknown search space. The main reason for this is unsuitability of exhaustive search for large search spaces (due to high computational costs). In some circumstances it pays to make a good guess instead of covering the entire search space. This is especially true for infinite rugged search spaces like in evolutionary computation. The EVMA provides a framework for program space exploration appropriate for multi-task distributed environments. We have applied the current architecture to a number of different computational problems (Chapter 9), and we believe the EVMA can be scaled to tackle real-life problems and can be applied to some complex computational problems subject to available resources. Further discussion on what exactly could be tackled by the current EVM system given a certain limitation of resources is provided in more detail in Chapter 9.

2.3 Virtual machine

The term virtual machine is generally used in three distinct ways. Virtual machine is used to denote:

1. a running, instantiated program (see the quote below);

2. an abstract specification for a computing device that can be implemented in different ways, in software and/or hardware;

3. a self-contained software/hardware operating environment that behaves as if it is a separate physical computing device.

The above definitions are not mutually exclusive. Definitions 2 and 3 can be treated as subclasses of the more general definition 1. Definition 2 mostly covers software virtualisation in the user space or application level (see, e.g., Pascal (Wirth, 1971), Lisp (McCarthy, 1960), Smalltalk (Goldberg et al., 1976), JVM (Venners, 1999), .Net (Miller and Ragsdale, 2003) virtual machine). Definition 3 covers virtualisation on the system level (system virtualisation implementations for operating systems, such as Xen, VMware, KVM).

Gelernter (1997, p.72) wrote (quoted below) about what is the relationship between a running program and a virtual machine:
A running program is often referred to as a virtual machine – a machine that doesn’t exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn’t merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.

This notion relies on the idea that logic and computation can be thought as being mechanical, they are abstractions of mechanical actions, therefore every running program can be thought as being a machine2.

2.3.1 Short history of the concept

The term Virtual Machine (VM) was created for the virtual machine environment (in the sense of definition 3) for the IBM System/360-67 mainframe in the late 1960s3.

Starting with the Intel 80386 in 1985, the x86 CPU architecture included hardware support for running multiple 16-bit DOS applications in the, so-called Virtual 8086 Mode. That is an operational state that allows a single x86 CPU to behave as if multiple CPUs were present. The Virtual 8086 Mode partitions the physical computing device into multiple virtual address spaces and maintains virtual registers for each of the virtual machines. Even though the concept of virtualisation was known since late 1960s, most of the support was done entirely in software. There was no hardware-based virtual machine mode for running multiple 32-bit operating systems until Intel announced its Virtualization Technology (VT-i) (Uhlig, Neiger, Rodgers, Santoni, Martins, Anderson, Bennett, Kagi, Leung, and Smith, 2005) in 2004 and AMD announced Pacifica (Ada, 2005) in 2005. These new advances in the hardware provided special instructions that facilitate hardware acceleration for the concept of virtualisation. In particular, they provided hardware support for fast context switching and hierarchical interrupt handling. This signifies the impact that virtualisation technology has and the general direction of computational architecture design.

It is possible to treat these advanced hardware technologies as a natural progression and reflection on the previously abundant software techniques. Each programming language, including the most rudimentary ones, provides the notion of context switching for method or function calls. There is

---

2“The virtual machine idea clarifies an important problem—what exactly do programmers do? What activity are they engaged in when they make a program? Are they technical writers? Are they mathematicians? Neither: They are machine designers. They need talent and training of the sort that makes for structural engineers, automobile designers, or (in a general way) architects—not for writers or mathematicians. A program is a blueprint for a virtual machine—a blueprint that gets converted into the thing itself (the executing program, the embodied virtual machine) automatically when you hand it to a computer.” (Gelernter, 1997, p.72)

3Virtual machines initially performed entirely in software, have been later provided through special hardware technologies – this provided faster and more robust partitioning between system images (Pugh, Johnson, and Palmer, 1991).
a hierarchy of execution contexts. In multi-threaded environments there are various complex dependencies of contexts and context manipulation is required. Contemporary hardware accelerated virtualisation solutions are nothing more than the extension of these basic concepts.

It is our goal to show that both the virtualisation and the notion of a virtual machine is useful, if not essential, in developing models of evolution, more precisely evolutionary computation. In the later chapters (Chapter 5 and 6) it is argued that current biological models of evolution involve complex interdependent networks of interacting entities. In this chapter we will look into possible architectures that would allow virtualisation to be used in evolutionary computation settings. The route to achieving this was to design an elementary prototype of the EVM in software, investigate various trade-offs and propose a refined software/hardware implementation that will become a basis for further studies into fully automatic virtual machine construction.

2.4 Design principles

Virtual machines (and, similarly, programming languages) can follow a number of possible organising principles. There are many trade-offs when designing a given computational architecture. The design goals generally can be classified into categories, such as:

- performance
- scope and provided functionality
- compactness of representation
- parsimony (simplicity) of the design

Some of the decisions we have made in the design and development of the EVMA are based on simplicity of the design (aesthetic choice); some are based on the compactness, efficiency and performance of the implementation. Most of the decisions, however, are based exclusively on the notion of evolvability, adaptability and the notion of meta-evolution (more detailed discussion follows in Chapters 6 and 8). The core idea of a hierarchically organised virtual machine model is the main driving architectural principle that has been followed. Most of the choices are geared towards providing robust and flexible architecture for an adaptable, evolvable and self-learning system.

---

4Execution context is the state of all computational units that participate in a given thread of execution. For example, an executing method will have certain local and global variables, their values or bindings, and the pointer to the current instruction that is being executed. When a method is interrupted and then resumed, all the states of this method need to be saved, and then loaded again. Context switching occurs when one method is interrupted and another is being resumed – one context is saved (for the interrupted method), and the other context is being restored (for the resumed method). On the CPU level, subject to a given architecture, the context is taken to mean the state of all the general purpose and state registers, together with stack pointer, flags and instruction pointer.
2.5 The EVM abstract architecture

2.5.1 Introduction

The EVMA combines elements of cellular automata and multi-agent systems in a unified model of a concurrent computational system. Both cellular automata and multi-agent systems can be viewed as two special cases of the EVMA. The EVMA is designed to be the superset of different computational models of distributed and interacting systems, with the goal of providing the most generic and abstract model to date.

The central part of the architecture is two-fold: a) on the lowest level, the EVMA uses a specially designed self-reflecting hierarchical computing language, called the EVM assembly; b) the EVMA facilitates communication and dependencies between asynchronously executing computational units called cells, that execute programs in the EVM assembly.

2.5.2 Abstraction

Abstraction is a tool for simplifying systems by focusing on subsets of relevant attributes and ignoring irrelevant ones. It also provides a mechanism for generalising properties that operate on multiple levels, capturing regularities and properties of certain processes. The EVM architecture facilitates the use of abstractions in computational models through special instructions capable of manipulating the machines and through mechanisms that operate across multiple cells. These constructions are discussed in more detail in Chapter 3.

2.5.3 Levels

The core notion of EVMA is the concept of the level. With analogy to levels of abstractions, computational levels form a hierarchical structure. Each level is characterised by certain encapsulated properties, notions, organisations, and patterns of execution. A given phenomenon can be described using various hierarchically organised descriptions based on several levels of abstraction. This decomposition follows certain organising principles (Haken, 1983), but in a general sense is arbitrary. A different perspective and different set of organising principles will construct a different hierarchical description and different levels of abstraction. As an analogy, computational decomposition into levels of computation is arbitrary and follows certain organising principles as guidance. The EVMA does not specify those organising principles, but rather provides a framework in which they can be externally introduced and used to generate a particular hierarchy of levels and abstractions.

An EVM level \( i \) is characterised by a set of computations that can be performed within that level on a single virtual machine \( M_i \). The level that the \( M_i \) computations are mapped to is referred to as the lower level machine and is usually represented by a lower index, \( M_{i-1} \). The level that \( M_i \) constructs at runtime (while being in a process of execution) is referred as the higher level machine, and is represented by a higher index, \( M_{i+1} \). This computational representation of levels is directly
related to hierarchical and conceptual modelling in general. The goal of the EVMA is to provide the ability to translate the usual conceptual representations of natural phenomena decompositions into the realms of (formal) computational hierarchical models. Such models then are amenable to empirical, theoretical and experimental studies. They can be changed, refined, observed and analysed with the use of computing machines; and also subjected to various experimental studies. The EVMA allows different decompositions to be compared side by side in a single unified hierarchical framework.

Due to the practical limitations of our computational implementation and due to the limits of physical implementations in general, each natural (e.g. biological) or computational system decomposition can be done only to a certain, arbitrary complexity level. Beyond the base, initial, non-decomposable level, the remaining levels (if there are any) must be collapsed and represented through probability distributions (usually). In other words, it is possible to postulate, but not practical to deal with, an infinite regress of base level decompositions. For any practical (and also theoretical) analysis, there is always something at the lowest level that is modelled implicitly through probability distributions.\(^5\)

A simple random search given a certain prior probability distribution can be seen as a simple concrete instantiation of the EVMA, where all the levels except the base level, have been collapsed into probability distributions. In this context, neo-Darwinism, as in (Huxley, 1942; Mayr and Provine, 1980), discussed in detail in Chapter 5, is thus a special case of the EVMA hierarchy, where all but the two levels: base and first meta-level, have been collapsed to a single probabilistic construct. In neo-Darwinism, all different identities of possible interacting machines have been collapsed to a single coherent view of an organism, or species, or kin – depending on which level a given neo-Darwinism theory is being applied to. Such a model can be characterised by simplified, single-direction causation.\(^6\) At this point however, the parallels of the EVMA and neo-Darwinian models end. The EVMA model is not, in general, limited to one or two levels. The EVMA allows for an arbitrary number of levels to be captured and represented explicitly. The EVMA represents a different, more general model of evolutionary systems, with a relativistic approach to causality and determinism. We will discuss these and related aspects later in this chapter.

### 2.5.4 Expressiveness

The EVMA supports the modelling of complex systems by employing highly distributed autonomous and adaptable systems that model naturally occurring phenomena, categorised generally as complex systems. An important aspect to note is that traditional virtual machines and programming

---

\(^5\)This is not anything new in itself. All disciplines, including evolutionary biology, always end up postulating a certain base level governed by a probability distribution (as in the case of random mutations). The new aspect lies in its translation into a proper computational model, and this is the main goal of the hierarchical aspect of EVMA.

\(^6\)The direction is arbitrary – in some models, like in gene-centric views of evolution (Fisher, 1930) popularised by the work of Dawkins (1976), it goes from genes to organisms and species. In the work of more traditional neo-Darwinists, it goes from an organism or species towards the genes.
languages are designed with human programmers in mind. This is not the case for EVMA. An automated search process in the space of computer programs could probably benefit from certain features that might be considered unproductive (i.e. confusing) for human programmers. However, these same features can be valuable for the automated program discovery process and automated program generation. Consider for example Dijkstra (1968)’s discussion of the goto statement. Dijkstra’s article suggested eliminating the goto statement because it decreased the ability of human programmers to analyse large quantities of cross-referencing code and to formally prove certain properties about the computation. However, in the same article Dijkstra (1968) argues that even though the goto statement is highly disruptive for human programmers, it is more than appropriate to leave the goto statement at the machine level, because of its high local expressiveness. Expressiveness in this context means that the goto statement can express the concept of changing an execution context in the most primitive and compact way. This high expressiveness can effectively lead to more efficient and faster running code, because the context change can occur anywhere and anytime. For automated programming systems, expressiveness for the operational context should be encouraged, even if the human programming facilities are being jeopardised. Expressiveness must take the automated program generation into consideration, not human level programming. This is the main principle.

To achieve these objectives we have designed the architecture primarily to be operationally expressive. That means, we enable various shortcuts and a compact representation for often complex computational constructs. The architecture should not only express simple things in a simple way, but it should also allow complex things to be expressed in simple ways, if desired.

Human-programmable aspects of the virtual machine were not taken into consideration (or formed secondary constraints). Formal analysis of expressiveness in computing languages and virtual machines is well presented by Okhotin (2005).

2.5.5 Incompleteness

The EVMA consists of a collection of independent computational units called cells. Each cell performs processing with respect to its own internal computational model. Cells have the ability to communicate with other cells in an asynchronous manner (not synchronised to any single global clock). Each cell can be thought as an interactive machine – see the discussion of interactive computation in Chapter 4.

Incompleteness influences the EVMA on two levels. On one hand, incompleteness is an inherent pragmatic property of a system with a high level of expressiveness. To be able to cope with bounded computation in an expressive computational framework, the cell must make choices with respect to limited (or specialised) sets of states and properties. On the other hand, due to the interactive and highly asynchronous nature of the computation, the overall behaviour of the system is hard (or impossible) to formally and completely express in terms of compact and human-readable algorithmic computation.
There are three main aspects of the abstract architecture that make it difficult for formal analysis. The first one is that the abstract EVM architecture can have an infinite number of cells, each of which can be refined an infinite amount of times. The second aspect lies in the fact that each cell can have a potentially infinite memory of past computations (its state), hence each individual cell and therefore the system overall is fundamentally history-dependant (or history-driven). The third aspect has to do with the self-reflective and self-modification capabilities of each individual cell. These aspects of the abstract architecture make complete formal analysis of the system behaviour and evolution difficult and perhaps unfeasible.

2.5.6 Hierarchy and cross-level linking

Although the EVM abstract architecture can be seen simply as a generalisation of a cellular system and a multi-agent system, it provides three important distinguishing features:

- an individual cell can be composed of many embedded cells (any single cell can be a composite cell);
- cells from many different levels may interact with each other (there is no strict partitioning of levels);
- the object of interactions are cells (i.e. computational units).

The aim of the abstract architecture is to provide a unified framework with the minimal number of postulated concepts at the conceptual level. There is no sharp distinction between the program and the data, no distinction between a cell and a computation, and any communication between computational units is also considered as computation (i.e. there is no distinction between computation such as addition and computation such as data transfer).

If one were to implement the architecture using a multi-agent system, that would mean that agents are structurally composed of micro-agents (for example as in the KEA/Opal multi-agent architecture (Nowostawski, Purvis, and Cranefield, 2001; Wang, Nowostawski, and Purvis, 2005). Although it is not common practice in multi-agent systems, in the EVMA it is possible for agents from multiple levels to interact with each other. We could also model messages as being potentially active agents in themselves (i.e. a message would be autonomous in the sense that it can perform certain activities autonomously).

If we were to use cellular automata, that would mean that an individual cell would be composed of an actual cellular automaton itself, with cells being able to interact across different levels with other cells. The messages between cells would be the actual state of cells too, eg. cellular automata (not the case in traditional CA) or the state of an individual cell (which is normally the case).

A discussion on the limits of physical reality and implications for computational models will be presented in Chapter 4.
All these frameworks above are equivalent in their modelling capabilities. They differ however from flat frameworks, such as classic CAs, in their ability to express levels of abstraction and a hierarchy of levels. Traditional CAs are unable to express certain cross-level interactions. For example it is not possible for a glider-gun or any other higher-order structure in the game-of-life CA (Gardner, 1970) to influence any of the underlying cells. The higher order constructs are (just) a direct result of the lower-level interactions and properties. The EVMA generalises that notion into bi-directional causation and an inter-level linked framework.

The EVMA is a general model that represents an attempt to capture the abstract capabilities of the computational hierarchy and cross-level linking. Other distributed models (e.g. cellular automata, multi-agent systems, Petri nets, etc.) can be viewed as special cases of the abstract architecture. They provide subsets of capabilities offered by the EVMA.

2.5.7 Constructions within the system

The EVMA postulates an interconnected web of cells. The concepts of tasks, rewards, environment and communication are provided uniformly through the cells or cell subgroups within the EVM Universe itself. These are all part of the EVMA architecture. There are no external entities outside the EVMA cellular system as such. In the abstract representation all of the communication, reward mechanisms, cell management and task management are performed by entities that are immersed in the EVM universe themselves. There is no intrinsic mechanism that is not or cannot be subjected to change.

It is our intention that the environment be ultimately implemented properly within the EVMA system itself. Due to practical limitations, simplicity of the prototype, and performance issues, the current computational implementation of the EVMA carries out communication, reward and feedback mechanisms, cell management and task management outside of the EVM universe. All of these mechanisms are collectively referred to as the environment. Without loss of generality, the environment is currently provided as an external set of mechanisms and we have investigated possible approaches for re-implementation within the EVM Universe itself. This will require a redesign of the actual cell computational architecture and the bytecode language. The currently tested models of environment are discussed later in Section 3.5. It is important to note that the notion of environment also encompasses the notion of the outside world. This means that physical world inputs are directly channelled into the EVM Universe through the environment. This notion, which we call an interactive computation model, will be discussed in detail in Chapter 4.

8One of the current limitations is the absence of compilers and interpreters from existing languages (such as C and Java, or Java bytecode) within the EVM assembly language. Once appropriate tools are provided, it will become easier to include the environment within the EVM universe itself.
2.6 Tasks

We use the terms problem and task interchangeably throughout this document. We do not take the formal definition used in theoretical computer science (where the meaning of the word problem is well-defined as a function on integers (Turing, 1936–7)). In algorithmic computation a problem refers to the fact that given some finite input, an effective procedure (Hopcroft and Ullman, 1979) will compute the corresponding output. If a problem is solvable, then its solution can be described by an algorithm, or an effective transformation of input to output that can be simulated on a Turing machine (a-machine) (Turing, 1936–7) (see also Chapter 4 for a detailed discussion and implications of algorithmic vs. non-algorithmic models of computation).

Our use of the terms problem and task is different – the concept of a problem is coupled with the learning mechanisms of the EVMA itself. By solving a task we mean a life-long continuous transformation of the input signals into output ones with the highest possible reward yield. Solving a task is an activity that a cell performs in a given environment. It means that the cell’s intake of rewards (Turing, 1948) is at its optimal level through the cell’s computational efforts. Therefore, solving a task is an asymptotic process of approaching the highest possible reward yield. A task is considered solved if the highest possible reward intake rate has been achieved for a cell. A task is not solved if the reward intake is below that limit. Note, that the abstract notion of a task may or may not have any direct representation within the EVMA environment. The task itself is an abstraction over various possible reward schemes of a given cell. The rewards that a cell obtains may come from a single target task or they may come from multiple environmental tasks that the cell addresses concurrently. A more detailed discussion of the reward schemes that are currently implemented and experimented with is provided in Chapters 3 and 9.

Note also, that although the task might be solved, and the optimal reward intake has been achieved, the actual computations carried out to yield this reward intake are not necessarily optimal in the traditional algorithmic and computational sense of the word. That means, that there might be other ways of yielding the maximum reward intake that are shorter, faster or in some other way more efficient than the given solution found. This is in analogy to traditional problem solving in computational spaces – a problem of sorting a list of elements may be solved in different ways, some of which perform better on a particular class of initial list orderings than others. Equally though, all of the sorting methods do eventually sort a list, and therefore are, in a computational sense, all equivalent (more on the notion of equivalence classes from the theory of computation can be found in (Hopcroft and Ullman, 1979)).

---

9Note that this definition captures the notion of solving a task, however, there is no effective procedure that can compute in advance what the highest possible reward yield actually is. Therefore, solving a task is always equivalent to an infinite computation of a Turing machine.
2.7 Cells

The concept of a cell is, beside that of levels, the second fundamental concept in EVMA. The cell in simple terms encapsulates certain behaviour (to put it precisely: computation). The internal representation of the cell as such is not specified. We use here the principle of encapsulation. In the current EVMI (see Section 3.4 for detail) the cell can encompass a single sequence of instructions, i.e. an EVM program, a machine with a program, or a complex search algorithm.

A cell reacts to its local interactions with its neighbours and with the environment. From a machine learning perspective, its goal is to solve a task. Computationally, this means that the approach the highest possible reward intake is the function of the cell’s own activities. This is, in general terms, achieved by adapting the cell’s activities to one or more reward mechanisms available in a given environment. We say metaphorically that the environment offers the cell tasks to be solved. In artificial life terms one can say that the cell aims to collect the maximal possible amount of food (or resources). It is similar to the notion of computational time and space units in the artificial life system Tierra (Ray, 1991a).

2.7.1 Intra-cell computing

Each individual cell encapsulates its own computation. A single cell (at the base level) corresponds to a single EVM program. By executing its instructions it manipulates its state, its internal virtual machine state and other virtual machines, on which further computations can be performed. Additionally, there are external mechanisms provided for input and output. The detail of these mechanisms are outside of the scope of EVMA, but will be discussed in the context of EVMI in Section 3.4.

2.7.2 Inter-cell computing

Cells are organised into a network. Each cell can be linked with a number of other cells. The links can be uni- or bi-directional. The links can be dynamically established and dynamically torn down by the cells themselves or by the environment. Networks can be further organised into higher-level networks. Any linked cells can communicate. The communication is based on transfer of information in the form of code and/or data between the connection ends. Put simply, a computation performed by a given cell (or network of cells) can be used (transferred) between the two connected ends. The requesting end is referred to as a requester and the providing end as a provider: a requester is provided a computation from a provider. Note, that this is different from pure information transfer between cells. In many traditional models of concurrent computing, the object of a communication is static information exchange. In the EVMA the object of the communication can be static information or it can be dynamic computation. Due to a unification of the program and data model, the program can be viewed as static information, too. But, this is just a special case of the generic scheme. The computation can be performed in the requester context (which is mostly the case in our experiments),

36
or in the provider context. The abstract architecture does not prescribe any implementation detail on the inter-cell computing.

There are a number of constraints that the practical implementation must follow. Some of the decisions that our practical implementation exhibits are due to the technological limitations of contemporary computing models and hardware constraints. We have also simplified some of the mechanisms, in order to conduct practical investigations on concrete aspects of the architecture.

Another property of inter-cell computation is the ability for an individual cell to merge with another cell, or a single cell to split into two independent cells. EVMA does not mandate any particular restrictions on these mechanisms. The detail of the implemented features for cell communication and merging and splitting are presented in Section 3.4.2.

### 2.8 Environment and cell topology

The EVMA is composed of cells only. The EVMA does not mandate any particular notion of environment, and does not even specify this concept. Within the EVMA everything is embedded within the EVM cellular system itself, and the notion of environment is not needed. However, for any practical use and implementation, some of the EVM cells act on behalf of external entities (outside of the modelled phenomenon or system), and therefore can be classified as external signals. Therefore, the external signals or inputs to the EVM system are simply provided as special cells. These cells (sensors and actuators) are called the environment within the EVMA. The EVMA does not specify or mandate any particular implementation or restrictions on the environment as such.

When discussing inter-cell communication, a central role is played by the actual topology of the network. The topology can dominate the dynamics in certain circumstances (see Janson, Luczak, and Rucinski (2000) and Albert (2001) for detailed investigations). In our present work we have concentrated on general properties, dynamics and structural tendencies that are independent of the actual topology. In most of the experimental work we have chosen arbitrary neighbourhoods and a topology that we felt was the most neutral for the given experimental setup. In general, in the context of the EVM abstract architecture, there is no particular constraint on the topology of the web of cells. In a global environment it can be \( n \)-dimensional and the neighbourhood can be as large as desired. For our preliminary experiments, we mostly used toroidal grids of cells with a von Neumann neighbourhood (4 neighbours: right, left, up, and down). However, for the extended EVM architecture with a 64-bit based addressing scheme, we used many individual machines linked into a hyper-cube-like topology. The EVMA leaves topological concerns to the EVMI, which means that the particular topology is not mandated on the EVMA level. Within the EVMA, the connections to associated cells can be arbitrary and physical proximity is not essential. The cell can keep the communication addresses of an arbitrary number of cells that are accessible for communication. Within the EVMA

\[ \text{\footnote{The initial studies and experimental results are discussed in Section 3.5.}} \]
this can be a number of arbitrary magnitude.

2.9 Complexity measure

The basic notion of complexity is adopted from the computational complexity measure, called algorithmic complexity, the Kolmorogov-Solomonoff-Chaitin complexity (Chaitin, 1987). Algorithmic complexity is a variant of information complexity applied in computer science that concerns itself with the relationship between computation and information. Algorithmic information theory principally studies algorithmic complexity and other complexity measures on strings and other data structures. Since most mathematical objects can be described in terms of strings, or as the limit of a sequence of strings, it can be used to study a wide variety of mathematical objects, including integers and real numbers.

There are several variants of algorithmic complexity, but we will use in this thesis the one which is based on self-delimiting programs (Li and Vitányi, 1997). By stating the complexity of a given self-delimiting program (or string) \( P \), we mean the length \( l \) of the shortest program that is capable of computing (in the sense of providing its output) \( P \). Algorithmic information theory gives formal definitions of a random string and a random infinite sequence that does not depend on physical or philosophical intuitions about non-determinism or likelihood. A random infinite sequence is one that cannot be computed by any finite program.

Some of the results of algorithmic information theory provide valuable insights into various models of computation. One of the interesting constructions is the definable but not algorithmically computable Chaitin constant, \( \Omega \). \( \Omega \) is a real number which expresses the probability that a random computer program on a Turing machine will eventually halt. Each single digit in the expansion of \( \Omega \) is non-computable and as such \( \Omega \) belongs to a class of random strings, often labelled as unknowable (Chaitin, 1999, 2005).

In the context of the EVM abstract architecture, the complexity is simply the total length of the shortest program that computes a given task. For practical reasons, we have decided to approximate tasks through the notion of finite tasks. By finite task we mean a task executed on a machine with an upper time constraint. All regular tasks are potentially infinite; thus the computation can go on forever, and the program constructing and executing such a virtual machine may be infinitely long. For the experimental work we use exclusively finite tasks and finite programs. This inherently leads to a separation between the EVMA and the EVMI (the actual implementation and experimental work). For some of the infinite computations (related to task solving), we try to maintain insights acquired from algorithmic complexity, and approximate certain tendencies and behaviour based on the results of experimental work with finite tasks. For example, the ideal program computing \( \Pi \) would run indefinitely, but, given its closed form we can discover it without waiting for an infinite amount of time. Even though \( \Pi \) represents an infinite object, the machine computing \( \Pi \) is finite and can be represented in closed symbolic form. Many useful infinite computations can be represented symbolically and then
be reasoned about using those symbols.

In the EVMA, multiple statically co-operating EVM cells can always be re-coded into a single EVM cell. Thus, a concrete execution of a cell program together with all its neighbours’ sub-routines, is conceptually equivalent to a single complex program that constructs, initialises and executes several distinct machines (assuming that the communication between the machines is conducted in a discrete and synchronised manner). In such a case the complexity measure discussed above continues to be applicable. This application of the underlying conception of this thesis must be qualified by the realisation that in general, the cells are autonomous and may process information in an asynchronous manner. That means that in practice, due to the autonomous property pertaining to the cells’ processing, the executing EVM Universe may not be reducible to a single cell and the standard measure of complexity will not applicable to the overall behaviour of the system. Note however, that from a single snapshot of a given functional mapping between the inputs and outputs of a cell (between a given start and end time markers), the complexity measure can be applied and it will be valuable and useful in certain circumstances. Thus, certain properties of a running system can still be investigated and analysed with the use of simple algorithmic complexity. For any full formal analysis of the general EVM Universe, the notions from current algorithmic complexity theory are of limited use.

In this chapter we have provided the general abstract description together with necessary formalisation and discussion of the properties and features of the EVMA. The next chapter contains specific implementation details and a description of a concrete instantiation of the EVM.
Chapter 3

EVM Instantiation
When doing mathematics, instead of burdening the brain with the repetitive job of redoing numerical operations which have already been done before, it's possible to save that brainpower for more important situations by using symbols, instead, to represent those numerical calculations.

Ernst Mach (1883).

3.1 Introduction

The EVMA model has been discussed in Chapter 2. In this chapter we present the concrete description of the Evolvable Virtual Machines Instantiation (EVMI). Given currently available computational architectures and programming platforms, implementing the computational model described in EVMA is not a simple task and can be carried out using different concrete instantiations. The number of instruction, redundancy and inter-connects between instructions are all dependent on a particular domain or particular experimental setup. EVMA is a generalisation of computation, in the same way as Forth Virtual Machine is. We will discuss shortly our preliminary or bootstrapping work, which will be followed by a discussion of various aspects of the EVMI implementation. Then we discuss the concept of task and the issues related to a task being solved (in abstract and concrete sense of the definition). This is followed by a discussion of the internal mechanisms and operation of individual computational cells and intra-cell communication primitives.

3.2 Bootstrapping implementation

The current EVMI provides a framework to experiment with and instantiate EVMA models. The current EVMI supports most but not all of the EVMA features\(^1\). The current implementation should be treated as a bootstrapping implementation that is used to investigate and provide a more refined base implementation. There is room for further refinement, and the actual concrete implemented features should not be taken as normative, but as steps towards the ultimate goal: an EVM system that closely follows the fundamental principles of the abstract architecture.

The implementation process started with many preliminary implementations, on different existing virtual machines. Originally, we sought to implement EVM directly on top of the Java Virtual Machine (Venners, 1999). However, the Java bytecode capabilities turned out to be too limiting in terms of reflective mechanisms. We tried Smalltalk (Goldberg et al., 1976), Lisp (McCarthy, 1960) and .NET virtual machines with their existing tools, language and bytecode support. These virtual machines provided enough capabilities in terms of tail-recursion and list manipulations, however, the representation and redefinition of individual instructions proved a major obstacle. One of the primary

\(^1\)The current EVMI is a refined version of earlier attempts that are not discussed in this thesis.
reasons to implement our own VM for the EVM bytecode is the ability of the VM to redefine its own base instruction set at runtime.

The final decision to design and implement our own virtual machine with its own bytecode language was the result of extensive theoretical and experimental studies of existing VM frameworks. Since the desired features were unique and could not be compromised, we could not reuse any of the existing virtual machines. As a result, there are several advantages of our own implementation, the main one being that we can develop a concrete implementations of the EVMA using different programming languages on different hardware without affecting the architecture as such. As a result the EVMI provides a general platform for investigating different architectural aspects of evolvability and computation on massively parallel asynchronous computing frameworks.

In the following sections we briefly present the current implementation, with some short discussion about the history of the development and the possible future extensions and research directions.

### 3.3 Tasks

In Section 2.6 of the EVMA we have specified the general abstractions with respect to the concept of tasks. The main problem with a concrete implementation is the fact that the solution to a given task is not known in advance, therefore, it is not possible to know (in principle) if a given task is solvable, and if so, when it has been solved. The EVMI therefore, inherently, must use certain computational approximations to the EVMA due to the physical limitations of current computational hardware used to implement the EVMA. There are different approaches addressing this problem. In this thesis we have used a scheme that is similar to reinforcement learning based on the Q-function. Detailed discussion of reinforcement learning concepts, and Q-learning in particular, is provided in Chapter 8. As in Q-learning (Mitchell, Utgoff, and Banerji, 1984, p.367–387), the actual real reinforcement mechanism is unknown and subject to dynamic changes. Basically, a computational entity, a cell, acts in an unknown environment, trying to maximise its own reward intake. The cell continues to adjust its actions in such a way as to collect more rewards and avoid punishments. The rewards (positive reinforcement) and punishments (negative reinforcement) can be delayed in time and do not necessarily correlate directly with the last actions of the cell. Solving a task is a form of life-long reinforcement learning for a cell, which continuously tries to obtain a positive balance of reward intake over the cell’s resource expenditure. In the EVMI we provide a certain initial amount of provisional reward for a cell. The cell computations consume the resources that the cell has been initiated with. As long as the cell’s reward mechanisms are refuelled and the cell continues to perform its activities, the task is considered solved within the EVMI context. Note, this is different from the ideal solution of the task in the EVMA. The EVMI approximates an unknown, infinitely long,.

---

2This is not different from traditional computational models based on Turing machines and their implementation on contemporary computing hardware: the abstract models are not concerned with physical limitations of the actual computing devices.
computational solution to the task by a temporal snapshot that has been computed up to a specified time mark. The task is considered as failed or unsolved when the activities of the cell do not bring the required reward intake for the cell, and cell’s computational expenditure cannot be balanced out with the cell’s reward intake. In our experiments the actual problems under investigation are of two types:

- there is a periodic feedback to the cell, with rewards, or punishments, or both (positive and negative reinforcement accordingly). The cell operates in an internal loop mode for an extended amount of time (for example see the maze experiments in Chapter 9). In this case the cell’s state trajectory is evolving in the state space in a more or less continuous fashion.

- there is only one reward feedback provided after all the activities of the cell have ceased (see for example the arithmetical experiments in Chapter 9). In this case the cell’s state trajectory has been clearly divided into generations. After each activity-reward feedback cycle, the cell’s state has been reset to the initial state. Note, that the environmental state is not being reset to its initial state, but continues its own evolution in a continuous fashion.

Thus, the ideal theoretical properties of solving a task by a cell within the EVMA are modified and approximated by the EVMI.

### 3.4 Cells

In our implementation, EVMI cells are constrained in such a way that an externally provided resource (reward) is necessary for the cells to continue to exist (or to be allowed to perform their activities). In other words the cells’ computational resources (memory and CPU cycles) are abstracted into a single parameter, called a resource, which again, is balanced by a single parameter reward. This is an external system-level constraint that may not have any direct linkage with the problem domain of any of the tasks. It can be treated simply as an organising principle (Haken, 1983). In other words, it is a domain-independent artifact that models certain constraints that are conceptually (metaphorically) equivalent to physical constraints. For example, two distinct material objects cannot occupy the same physical space, or a material object cannot emit energy indefinitely, etc. Such physical constraints and more complex organising principles are modelled through the abstract notions of a resource and reward.

#### 3.4.1 Intra-cell computing

In the EVMI we have used a collection of cells with a single EVM program. The cell program takes data from and produces output to the environment (discussed in Section 3.5). In the abstract architecture these mechanisms can be arbitrary, and the EVMA does not mandate any particular mechanism. These mechanisms and any other search mechanisms are not part of the individual EVMA cell’s architecture themselves and, as with any other cells, could be changed without compromising the
overall EVMA. However, due to high computational costs, experimental studies with high levels of flexibility were not feasible on the available hardware. Therefore for efficiency reasons we have primarily designed and used certain fixed global properties of the cell processing. Thus, though each cell specialisation and program execution mechanism was built-in, and could not be changed by a given cell itself, in the abstract sense, these could have been reified and provided as mechanisms on the base level, subject to direct manipulation, degradation and improvement when needed.

3.4.2 Cell operation

There are some differences between the abstract architecture and the actual prototype implementation used for the experiments described in this thesis due to specific features used for the cell implementation. In EVMI implementation every cell continuously iterates through the steps of selecting a program, running it and, depending on the rewards, modifying the program structure.

At every iteration, the execution engine tries to solve some tasks provided by the environment. For each task, it runs the cell’s program (possibly calling some neighbours’ programs) in interaction with the task’s resource. If the output is correct, the cell receives some rewards. A specialisation mechanism uses these rewards to modify its state and provide a new program for the next iterative step. The specialisation mechanism is an umbrella term that relates to the ability of the cell to trim its computational capabilities (that means some of the computational capabilities are removed from the cell repertoire). Generally, each cell starts the search process as a general computing machine, capable of computing any computable function. This capability however is not needed for a particular one-off task – what a cell needs is only a subset of the general capabilities. In other EC systems the decision to trim the capabilities is done outside of the search process through the appropriate selection of machine instructions used. This is augmented in EVMI by the mechanisms that allow a cell to trim the instruction set during the runtime too. This trimming process we generally refer to as specialisation. The general instruction set is being trimmed and specialised for a narrow task or purpose.

Individual cells work in a distributed and asynchronous fashion. The processing of an individual cell is sequential and consists of sets of operations, also called iterations. The iterations can be imposed by the environment (as was the case with some of our experiments), can be managed by the cell and cell program alone, or they can be in an arbitrary order (subject to a mixture of environmental and cellular influences).

At every iteration, the specialisation mechanism of the cell must generate one program. This program will try to solve some tasks from the environment, and this process may yield some rewards coming from the environment. From an artificial life perspective, the program attempts to maximise the reward and maintain its existence by trying to achieve homeostasis with regard to the external resources (rewards). The cell receives rewards according to the performance of the program. The

3Often, in this thesis we use intentionality when describing program or cell behaviour. It must be remembered that this
specialisation mechanism uses these rewards to modify the program.

A key feature of the cellular systems is the interactions among their basic components: the cells. From these interactions emerges the global behaviour of the system. We believe that symbiosis among the cells is a crucial property to reach higher levels of complexity. Also, from a machine learning perspective, symbiosis facilitates knowledge diffusion and reuse. Knowledge reuse, on the other hand, facilitates multitask learning. Any cell can use any other cell’s program (as long as they are connected) to solve a given task. In other words, programs of a machine can use neighbours’ programs (linked cells) as their own instructions (Figures 3.1 and 3.3). For example, a program can look like the following: perform addition, duplication, left-neighbour-program, multiplication, right-neighbour-program:

```
add
dup
leftNeighbourProgram
mul
rightNeighbourProgram
```

If rewards are gained and these rewards are shared between all of the involved cells so that they all benefit from the relationship, we can model this as a simple symbiotic mechanism based on mutualism (as discussed in Chapter 5). As a consequence, two (or more) cells can effectively merge (operate as a single organisational unit), in the reward-driven dynamics. The reward sharing is distributed proportionally to all the merged cells. Detailed discussion about this phenomenon is provided in Chapter 5.

### 3.4.3 Data passing

One of the popular models for implementing cellular interaction is based on the use of special input/output instructions to pass data (messages), or to use the actual state of the cells as a communication medium (typical in cellular automata). In the case of communication channels with read/write instructions, these instructions respectively write to, or read from, a data structure – typically a buffer or a register.

In the EVMI, read/write interactions would substantially complicate cell-to-cell communication, as has been discussed in detail in the system Phuon (Capcarrère, 2002, 2004). For example, suppose intentionality is an externally imposed abstraction over completely unintentional computational processes that control the behaviour of a program or cell. In terms of trying to achieve homeostasis, the homeostasis is either achieved or not, and the cells that achieve it persist in the system and those that fail are removed and replaced by other cells. Therefore, it appears as if the cells are trying to achieve homeostasis, where in fact they merely follow their own deterministic computational trajectories. The use of intentionality and causality helps in discussing and explaining the dynamics of the system and we use it throughout this work, but the reader should keep in mind that all emergent properties are outcomes of statistical interactions on lower levels, and the cells are not rational or causal agents in their own right.
a cell $c_1$ writes some input in a neighbour $c_2$’s buffer. $c_1$ usually expects $c_2$ to write the result back in $c_1$’s buffer. When does $c_1$ know it can read the result of $c_2$’s computation from its buffer? How can $c_1$ be sure that the result was written by $c_2$, and not $c_3$? These synchronisation issues render cell communication complicated and can create deadlock issues. Complex interrupt handlers need to be employed to address issues related to asynchronous communication. These are undesirable artifacts of a simple traditional read/write interaction model.

The EVMA does not mandate any particular implementation for data passing. The currently tested implementations did not use any of the explicit read/write constructs, with the interrupt handlers, that would be the most efficient with contemporary hardware systems. It is nevertheless relatively straightforward to implement communication in various ways. Since the read/write models are complex due to synchronisation issues, we have opted to avoid such a model for our EVM implementation. The functionality of such a model however can be simulated by the EVM implementation, based on the notion of cell merging – in this case data passing from one machine to another is implicit in the structure of the sequence of executed instructions. Therefore read/write operations can be simulated without explicit read/write operations.

Other cellular systems, like classical cellular automata, seem to have behavioural properties that are sensitive to synchronisation issues. Indeed, some authors argue that some of the apparent self-organisation of CAs is an artifact of the synchronisation of the global CA clock (Sipper and Tomassini, 1998). Others claim that information is less easily propagated in asynchronous CAs and that such CAs may be harder to analyse (Wolfram, 2002). That would be also the case with an asynchronously executing and communicating multi-cell EVM system. With the current implementation, even though individual cells work concurrently, each cell’s own view of the execution is synchronised and straightforward to track and analyse. Each cell operates independently of the other, in an asynchronous fashion. However, each individual cell represents only a single thread of computation and is therefore traceable and easier to analyse. In principle, in the abstract architecture, each individual cell may be composed of many asynchronously executing cells. Therefore, each individual cell would be capable of modelling a parallel distributed system on its own, keeping detailed track of all the executed sequences and data. Even though it is possible to show that this is feasible in principle, it would complicate the storage and execution for practical implementations at the present time.

### 3.4.4 Cell merging

There are two basic mechanisms to implement simple merging in a cellular system. One way is to have a single program per cell and provide a mechanism to merge individual cells into higher-order cells (collaboration). The other possibility is to provide the architectural mechanisms for a cell to have the ability to contain multiple programs (containment). We have implemented and tried both of these variants.

In the case where a single cell is capable of containing several programs (we also refer to such
a collection of programs as a *machine*), our primary goal was to specialise a single cell for solving a set of tasks. It is a basic model for multi-task learning – several programs per cell together with a decision mechanism within each cell which decides when to use which routine. It is easier to have several programs (or several building blocks) solving certain specialised tasks and having a management layer that can decide which program is appropriate and to be used for what situation. In the extreme case, we could imagine having as many programs per cell as tasks in the environment and a good discrimination mechanism. In the experimental section we will discuss some issues regarding the co-evolution of a cell’s programs.

With only one program per cell, locality of interconnections between the cells plays an important role, and our EVM cellular system exhibits interesting artificial life features and structural tendencies. For instance, since knowledge sharing is not as straightforward as with several programs per machine, the system must find some artifacts to propagate solutions to other cells that need it (e.g. through parasitism, which is discussed later in Section 9.6). Moreover, because each cell remains quite simple (just one sequential stack-based program), the implementation of the specialisation mechanism is facilitated more easily (it has to specialise only one program). This is especially well-suited for genetic algorithms and probabilistic search methods. Thus because of its simplicity and generality, in most experiments we chose to use a single program per cell model.

### 3.5 Environment and resources

Every cell maintains a program. The cell’s *goal* is to find a successful program: one that, by solving a task, yields enough rewards for the cell to survive. Programs can call other programs (Figure 3.1 and 3.3).

When a given program obtains a reward, the reward will be shared proportionally with any programs used as *assistants* to compute the solution. All of the participants will benefit from their relationship. In other words, symbiotic (mutualistic or parasitic) relationships will appear between programs. This ability to access other programs has thus facilitated complex hierarchical organisation and self-assembly. As a consequence, cells are able to collaborate to solve complex problems. Problems that none of the cells would be able to solve on their own can thereby be solved through cell collaboration.

Figure 3.1 depicts a simple example of cell $C_1$ using other cells during the course of $C_1$’s computation. The computation starts with $C_1$ executing its own instruction number 1. The second instruction

---

4 For example, for our stochastic search methods we had to choose the number of programs per cell. That was not intuitively straightforward, as the environment is dynamic: new tasks may be added or removed. Therefore, certain simplifications must be made that violate the abstract architecture to a certain degree.

5 In the case of multiple programs per cell, programs naturally interlink and use each other within a cell to solve more complex tasks. With only a single program in a cell, the information and capabilities sharing must be conducted across the cellular level.

6 A program in this case means simply a sequence of instructions in the EVM assembly language.
of \( C_1 \) calls its right neighbour, the cell \( C_3 \). Therefore the computation follows to the first instruction of the cell \( C_3 \). \( C_3 \)'s program is executed up to instruction number 2, and the third instruction of \( C_3 \) calls the program of upper neighbour, cell \( C_5 \). And so on. On the bottom of the figure you may see the actual order of instructions being executed and where they belong across all cells. This type of hierarchical assembly between the cells is distinct from the hierarchy on the virtual machine level, which is supported by the EVM assembly language. This multi-level machine hierarchy (as opposed to cell’s hierarchy) is being depicted on Figure 3.2. Note that the Current Machine (CM) index and Higher Machine (HM) index may point to various parts of the HM list.

Figure 3.1: The dark cell executes its program. Arrows show instructions that call neighbours’ programs.

The environment represents the external constraints on our system. Its role is to keep the system
under pressure to force it to solve the tasks specified from the outside. We have designed the environment as a set of resources. There is a one-to-one mapping between the resources and the tasks to solve (every resource corresponds to a task). The purpose of these resources is to give rewards to the cells when they solve their task.

Cell specialisation consists of finding a successful program for the cell. In other words, the cell self-adapts to a particular task in the environment. Several specialisation mechanisms have been studied by the author: classic genetic algorithms, ad hoc stochastic search (maintaining a tree of probabilities of potential building blocks), or an adaptation of an environment-independent reinforcement learning method (proposed by Schmidhuber (1995)). We have experimented with different methods of learning, including random search, stochastic search and genetic algorithms. The results of our experiments have been published by Nowostawski, Epiney, and Purvis (2005) and are discussed in Chapter 9.

Some argue that many multi-task problems share a similar internal structure, therefore reusing common properties and regularities aids the solution search process (Thrun, 1996). Multi-task learning is an area of machine learning which studies methods that can take advantage of previously learnt knowledge by generalising and reusing it while solving a set of possibly related tasks (Baxter, 2000). Our EVM architecture benefits from those properties, by concurrently solving multiple tasks and by recombining the already obtained partial solutions or solutions to sub-problems.
Figure 3.3: Instruction set $I$ for the dark cell’s program. Instructions 21 to 24 execute the programs of its neighbours. The von Neumann neighbourhood (on a regular grid: up, right, down, left neighbours) is used for this example. A program can thus look like the following: $\text{add dup leftNeighbourProgram mul rightNeighbourProgram}$. 

### 3.6 The EVM implementation

#### 3.6.1 The EVM assembly language

There exist many programming languages developed within the field of Evolutionary Computation (EC). Usually EC employs higher level programming languages that are designed for human programmers (such as Lisp for the original formulation of tree-based Genetic Programming (Koza, 1992)); some are developed with an evolutionary process in mind (Spector and Robinson, 2002; Ryan et al., 1998) and others are developed for machine processing and recursive program manipulations (Schmidhuber, 2004). Some of the languages are highly specialised and provide the evolutionary mechanisms with a bias toward a particular solution subspace. However, none of these languages provides mechanisms to arbitrarily manipulate levels – a property needed for our EVM implementation. There are other features we want our base machine language to possess, which none of the existing languages have. For example, we want the language to be capable of redefining itself. That is, the primitive instruction set must allow the evolutionary process to restructure and redefine itself. Also, we want a programming language that is highly expressive: solution programs to typically encountered tasks must be compact. We believe, based on the work in related fields (Schmidhuber, 2004) that there are efficiency advantages for a language whose solution spaces may frequently be highly recursive.

A programming language used for search in EC plays an important role. Each given programming language is particularly suited only for some problems. One of the appealing aspects of a multi-level search process is that it can, in principle, define a new base level and a completely new

---

7 Search means here enumerating through a program space and comparing a given program with the desired one. Solving a search problem in EC means finding a program that performs a given assumed functional mapping, or, as in the case of EVM, performs a particular computation (finite or infinite in nature).
programming language that is specialised for a given task at hand. We wanted to design the EVM system in such a way so as to be able to exploit this property.

The instruction set contains a set of standard bytecode-like instructions, similar to LISP or Java bytecode. We also have special instructions to manipulate levels: to create a level and to navigate between levels. This is a novel feature that distinguishes our architecture from existing virtual machine implementations.

3.6.2 Integrated EVM environment

Our Java implementation provides the core EVM engine, that includes an EVM bytecode interpreter, a mnemonic compiler, a disassembler and a pretty-printer. These tools can be used independently of the rest of the EVM Integrated Development Environment (IDE). The IDE provides a comprehensive GUI environment, with many features for customisation, monitoring and simulation of different experimental scenarios (see Figure 3.4). The EVM IDE is developed in Java, and therefore can be used on any platform that is capable of executing Java 2 (JDK 1.5 or later). The software is open source and is available for download from the Cirrus project on SourceForge (http://www.sf.net/projects/cirrus). The Java implementation can encounter substantial performance issues with larger search spaces. To address that problem, we have re-implemented the core of the system in C and assembly language, to take advantage of the existing hardware support. The sources for the C implementation for various platforms are also available from the above SourceForge project.

Figure 3.4 depicts an example run of arithmetic simulations. On the upper left corner we can see a snapshot of the grid with cells represented as rectangles and colour coded function units they currently obtain their rewards from. Below that window we can see a dialog box for specifying some initial system configuration, tasks, the grid size, etc. On the right hand side we see the details of the programs and the EVM assembly instructions with some verbose output about the system state. This is provided as textual information, however, on the bottom of the diagram we can see some statistical information presented in graph form – in this particular example we depict the probability distribution for various instructions as a function of time (iterations) performed by the system. The individual cell states can be inspected by clicking on the cells themselves causing the window in the middle (foreground) to be displayed. This shows the exact instructions together with their probabilities (colour coded) and the freezing factor (discussed in detail in Chapter 9).

3.6.3 General information

As stated earlier, the main objectives of our computing model are to provide the following features:

- a simple threaded model of computation, where the multi-threaded and multi-task aspects are decoupled from the core computation models,
Figure 3.4: Screenshot of the EVM IDE. Cell customisation, graphs generation, online evaluation, monitoring, and visualisation can be easily performed with few mouse clicks.
- a flexible reification model, where all computing aspects, data structures and control flow can be reified and re-defined,

- an expressive language, which is capable of expressing complex processing instructions in a compact fashion,

- the ability to utilise self-organisation and other mechanisms that boost evolvability of the computing model,

- the provision of a flexible and extensible framework for the study of multi-task learning and adaptive systems,

- the provision of a framework for the study of massively multi-threaded computations.

The computing machine metaphor at the base level is similar to the original Turing machine. The notions of instruction, execution and storage elements are similar to any other imperative computing model and Turing machines in particular. Similar to other low-level virtual machines, there is a concept of data storage (through stack and list data structures) and a concept of an instruction with arguments, performing a certain unitary operation on the data structures. In the implementation all data and instructions are represented uniformly as integers.

### 3.6.4 Stack and register machines

Our implementation of the EVM architecture is based on a stack machine. The central element of any stack machine (not surprisingly) is a stack. A stack is a simple linear data structure capable of storing data elements. The elements can be pushed onto the top of the stack, and popped from the top of the stack. It represents first-in-last-out (FILO) data flow. Apart from pushing and popping data elements onto and from a stack, there can be other stack-manipulating instructions available, such as swapping the order of the top two elements or, in the general case, \( n \) elements of the stack. The most important feature of the stack machine is the fact that all processing operations take their input from and provide their output to the stack. We say that all operation operands are passed through the stack. We will discuss the detail of all the stack-manipulation instructions later (Section 3.6.9).

Historically, virtual machines were designed and implemented as register-based machines (Blunden, 2002). Such a register-based design maps more directly to a particular hardware implementation. All mainstream CPU designs are based on the notion of registers and these CPUs are well suited for register-based computations. The biggest strength of register-based machines is the fact that they are straightforward to implement, both in hardware and software. However a register-based virtual machine is constrained to a particular register-based hardware implementation and it can be difficult to port it easily to various, often incompatible hardware implementations. Hence, for added flexibility and robustness within a virtual machine, there is a tendency to make the design independent of the underlying implementation. One way to achieve that is to implement a register based VM in such a
way that it can be ported to a multitude of various register-based hardware. Another approach is to use stack-based machines.

Stack-based virtual machines (Koopman, 1989) do not pass arguments (operands) in the body of the program. There is a special data structure, a stack, that is used to take the operands from and to insert the results on Stack based VMs pop and push operands and results to and from the stack. Consider the following example of a typical register-based assembly code to add two numbers together:

```
mov $1, %r1 ; place argument 1 into register r1
mov $2, %r2 ; place argument 2 into register r2
add %r1, %r2, %r1 ; add the value of r1 and r2
;and place the result in r1
```

This is encoded for a stack-based VM as follows:

```
push $1 ; put 1 on the top of the stack
push $2 ; put 2 on the top of the stack
add ; pop two elements from the stack, add them,
     ; and place the result on the top of the stack
```

This example demonstrates the strengths and general properties of stack-based and register-based virtual machines. Register-based virtual machines are capable of carrying on multiple computations with partial results stored in multiple registers simultaneously. In contrast, stack-based machines are well suited to carry out a single thread (a single flow) of computation, that processes data sequentially. A single flow of control in a linear fashion. Note, register-based and stack-based machines are equivalent in their computing power and can be implemented interchangeably with one another. However, the point is that for single-threaded, single computations with linear characteristics, stack-based virtual machines are easier to program. The code is much more compact, because there is only one common data structure, and it does not require addressing. Unlike multiple registers that always require appropriate discrimination within the body of a program, stack-based programs require fewer arguments and tend to be more compact and expressive. Of course, in complex multi-threaded or multi-objective computations, stack-based machines suffer due to the number of stack manipulations necessary to keep all the partial results ordered appropriately on the stack. In a naive and simplified way one can view register-based machines as a generalisation of stack-based machines over multiple

---

8 Note, it is possible to have stack-based hardware CPUs too. There is an interesting research area regarding different CPU designs and often it is research and development of the software virtual machine that stimulates hardware architectures and developments (Koopman, 1989; Fox, 2002).

9 We use here a pseudo assembly code. The register number for the SPARC architecture can be between 1..32. Register %r0 always holds value 0.

10 One can have a stack-based VM running on top of a register-based VM. In the case of our EVMI, the base EVM is stack-based and runs on top of stack-based JVM, which runs on register-based hardware.
stacks with stacks limited to a single word. The EVM architecture is another such generalisation. In the EVM, there are cells (independent stack-based processors) linked up together to allow multi-threaded multi-objective computation. For that reason, for a single individual cell, a single-threaded single-line-of-control computations seemed more appropriate. This is because a single cell is designed to deal with a single control flow and single sequence of execution, for which stack-based processing is particularly well-suited.

3.6.5 The base EVM stack machine

With small differences, the base EVM is comparable to an integer-based subset of the Java Virtual Machine (JVM). The basic data unit for processing is a 64-bit signed integer\(^{11}\). The basic input/output and argument-passing capabilities are provided by the operand stack, called the data stack, which is just an ordinary integer stack. The operands for all the instructions are passed via the stack. The only exception is the instruction push, which takes its operand from the program body itself.

3.6.6 Base Machine instructions

In the following sections we will present the instructions that comprise the base level virtual machine. Each individual instruction is implemented as a Java class, with the class name prefixed with Op followed by the mnemonic of a given operation, for example nop is implemented by OpNop Java class. The assembly interpreter matches the mnemonic with an appropriate implementation class and passes the execution to this class. This provides a plug-able, extendable and flexible architecture. Adding new primitive instructions is simple, it is achieved by the user implementing self-contained code in an appropriate class. The architecture benefits from the object-oriented Java model with the use of encapsulation, polymorphism and object-oriented design patterns. However, this high level approach suffers in terms of performance. The matching of individual instructions to the appropriate classes and delegation of execution are expensive in terms of execution time. As a consequence, this implementation model was changed for the C implementation, which uses pointers to appropriate functions and stores them in indexed arrays. There are substantial performance benefits in dispatching the instructions directly using fixed pointers and array-based data-structures. But flexibility and maintainability of the implementation is traded off for higher performance.

3.6.7 Indexing the instructions

In traditional stack-based VMs designed for human programmers, there are two cases which should never occur, and it is the compiler’s duty to generate correct code and the runtime system’s job to ensure that these two situations will not occur. The first situation is when a given index of a list

\(^{11}\)This somewhat arbitrary constraint is dictated by the need for efficient implementation on contemporary computing devices.
(or index of the instruction to be fetched) is out of bounds. We will discuss this case below in this section. The second case is when the operand stack does not have a sufficient number of parameters for a given instruction to execute correctly. This case is also discussed in the section below.

In the EVM implementation, all the instructions\textsuperscript{12} are indexed by an appropriate integer. Thus, if a given level has \( N \) instructions, there is a notion of a list of instructions that contains \( N \) elements (instructions). Those instructions are indexed by appropriate instruction code (binary code for EVM), then fetched, then executed. Normally, the indices must be within the proper range, that is, it is an error to address instructions below the starting index or above the list length. To provide a straightforward mechanism such that an arbitrary integer used as an index is always valid, we simply use the so called modulo rule. The general modulo rule is that if the indexing argument is out of range, the implicit operation, modulo the length of the list that is being indexed, is performed. For example, consider the operation pick that takes its argument \( i \) from the top of the stack and then picks the \( i \)-th element from the stack. Let us consider the case where there are the following elements on the data stack (from the bottom of the stack): \( a, b, 3 \). The argument for the pick operation is 3. However, there are only two data elements on the stack, indexed from 0, left on the stack after the operand 3 was picked. Thus, the operand 3 is converted with the use of the modulo rule, and the effective index is calculated as: \( 3 \mod 2 = 1 \), then and 1 will be used as an effective index. Index 1 would point to the top stack element \( b \). In another case, if the initial stack was \( a, b, 2 \), we would have \( 2 \mod 2 = 0 \), and the element indexed 0 would be \( a \).

3.6.8 Execution errors

Sometimes the modulo rule will fail to provide a correct execution. The typical case is when an operation requires a certain number of operands on the stack, but there are none, or an insufficient number. Consider the case of a pick operation with an empty stack, or a stack with only a single value, the argument for pick opcode (pick requires at least two elements on the stack: the indexing argument and at least one data argument). There would be no values left on the data stack, therefore the correct execution of the pick operation would be impossible. The EVM machine has a Boolean flag that indicates the default behaviour. This flag is called COE, which stands for Continue On Error. The operation (opcode) that sets the COE flag to true is called coe. If the flag is set to true the execution will simply ignore the error, undo the partial execution of the faulty instruction, and continue the execution of the program by executing the next instruction. That means, the faulty instruction will be simply ignored and will have no effect on the computation state. In case COE is set to false, the execution frame will halt the execution of the current program if any error occurs. There are two primitive instructions to control the COE flag. The instruction coe will set the COE flag to true, and hoe will set it to false. Hoe stands for Halt On Error.

In both of the cases discussed above, a traditional VM (e.g. the JVM) would raise an exception

\textsuperscript{12}Throughout this thesis we use the term operations to denote instructions on the base EVM level.
at runtime, effectively halting the entire program and all the execution frames. Those are considered serious and non-recoverable runtime errors. In the case of the EVM, these are part of the normal semantics and the “erroneous behaviour” itself can be effectively used for programming, that is, achieving certain computational states. For example, faulty instructions (that are not executed) are there in the program only as markers, or instructions are executed (expressed) only in certain situations and are inhibited in other situations (there is an analogy with biological gene expressions). The modulo rule and coe instruction allow any randomly generated code to execute on the EVM base machine with consistent semantics. Note, however, that these features depart from traditional virtual machines (such as the JVM) and are typically not suitable for human programmers to take advantage of. It would normally be unsuitable to use humans to program such virtual machine. The process would be difficult to trace, debug and it would be difficult for humans to obtain predictable behaviour due to the extremely large program space. In traditional virtual machines, that program space is significantly reduced through various constraints and error checking (both at compile and runtime). In the case of the EVM, the program space is left open and unconstrained. This is because automated program generation tools can navigate and traverse such an unconstrained search space in a more efficient way, taking advantage of various usable substructures. Automated and adaptive search algorithms (such as assorted evolutionary computation methods) searching in traditional program search spaces have difficulty traversing the search space due to considerable discontinuity in the search landscape. With the EVM search landscape this issue is (at least partially) addressed.

### 3.6.9 Data stack

In the following sections we will discuss individual operations (opcodes) of the EVM base machine that relate to operations on the operand stack. The operation and overall computing model are discussed below. We include the number of input and output operands that are taken from and placed on the stack, in the form (input $\rightarrow$ output). The appropriate part is left out if no operands are taken from or placed on the stack. Numerals are represented with a preceding dollar sign, and variables are represented with lower case characters.

**nop**

*nop* does not do anything. It is an empty operation, without any side effects. It occupies one position in the program body and can be used as a blank filler or label for jumps.

**const\_1 $\rightarrow$ $\$1$$

*const\_1* places a value 1 on top of the data stack. Similarly, operations *const\_2*, *const\_3*, *const\_0*, *const\_m1*, and *const\_m2* place 2, 3, 0, -1, -2 on the stack, respectively.
**push**

*push* is a unique instruction in the instruction set. All instructions always take their arguments from the operand stack. *push* however, takes the next instruction in the program list (treated as an integer) as its argument. It pushes the next integer on to the top of the stack and then skips the next instruction execution and continues with the instruction following that.

**pop** \((a \rightarrow)\)

*pop* pops the top element from the stack. The element is simply taken from the top of the stack and disposed of.

**popn** \((n, a_1, \ldots, a_n \rightarrow)\)

*popn* is a generalisation of the *pop* operation. *popn* takes one argument \(n\) from the stack and then takes \(n\) additional elements from the stack.

**swap** \((a, b \rightarrow b, a)\)

*swap* swaps the order of the top two elements of the data stack.

**roll** \((n, a_1, \ldots, a_n \rightarrow a_n, \ldots, a_1)\)

*roll* is a generalisation of the *swap* operation. It takes \(n\) elements from the data stack and permutes their order (back to front). This operation could be called *swapn* in consistency with other \(*n\) operations, such as *dupn*, but, for historical reasons, we use the original *roll* name, from the Forth implementation (Moore and Leach, 1970).

**dup** \((a \rightarrow a, a)\)

*dup* duplicates the top element on the data stack.

**dupn** \((n, a_1, \ldots, a_n \rightarrow a_1, \ldots, a_n, a_1, \ldots, a_n)\)

*dupn* duplicates \(n\) top elements on the data stack.

**depth** \((\rightarrow depth)\)

*depth* pushes the current size of the stack onto the stack.

**pick** \((n \rightarrow a_n)\)

*pick* picks the \(n\)th element from the bottom of the stack and pushes it on to the top of the stack. The elements are indexed 0 onwards from the bottom of the stack.
plength ($\rightarrow P_{\text{length}}$)

Pushes the length of the current program onto the operand stack.

### 3.6.10 Lists

Unlike other virtual machines (e.g. the Java Virtual Machine), the EVM base virtual machine does not provide any operations for creating and manipulating arrays. Instead, the architecture facilitates operations on lists (in similar fashion to LISP or Scheme virtual machines). There is a special stack, called the list stack, or l-stack for short. The l-stack can store integer-based lists. This l-stack data structure is an a-stack, that is the top element of the l-stack is kept in a separate placeholder, called simply a list.

The following set of operations operate on the list, i.e. the topmost element of the l-stack.

popn_l ($n, a_1, ..., a_n \rightarrow$)

popn_l pops $n$ top elements from the operand stack and puts them into the list, on top of the list stack. If $n$ is bigger than the current stack depth, the number of elements to be popped is calculated as $n \mod \text{depth}$.

To explain more explicitly what is happening, we will use the notation that is used in the EVMI specification in the user (and JavaDocs API) documentation. This notation consists of a presentation of affected data structures before and after the execution of a given instruction. In the description below, $n$ elements are present on the operand stack, and are removed after the instruction is executed (we refer to these configurations, as “stack before” and “stack after”, respectively). We use similar notation for the list and l-stack data-structures. The list before execution of the instruction consists of a list $l_0$ with elements $< ... >$. After the execution, all the elements $< a_1, ..., a_n >$ originally from the stack are used to form a list $l_a$. The list $l_a$ is placed into the list data-structure, and pushes the current list $l_0$ onto the top of the actual l-stack stack. We schematically represent this using the following notation:

stack before: $n, a_1, ..., a_n, ...
stack after: ...
list before: $l_0 <...>
list after: $l_a <a_1, ..., a_n>
l-stack before: $l_1 <...>, ...
l-stack after: $l_0 <...>, l_1 <...>, ...

In this notation we depict the state of the data stack, list and l-stack before the operation and after. The lists are tagged with l_number pattern, followed by the actual list elements in the <> brackets. The stack lists elements, from left to right, that represent from the top of the stack, to the bottom.
Same for the l-stack – the elements are simply listed from left to right, and from the top of the l-stack to the bottom. Non relevant items are shown as dots. The data structure that is not affected by the operation is omitted in this notation.

**prepend**

Prepends the value from the data stack to the list.

- stack before: v, ...
- stack after: ..., 
- list before: 1_0 <...>
- list after: 1_0 <v, ...>

**append**

Appends the value from the data stack to the end of the list.

- stack before: v, ...
- stack after: ..., 
- list before: 1_0 <...>
- list after: 1_0 <..., v>

**load**

load pushes the value i from the data stack. Then it pushes the value indexed by i (the list’s ith element) from the list, on to the top of the data stack. In other words, the top element of the stack (namely i) is replaced by the copy of the ith element of the list.

- stack before: i, ...
- stack after: a_i, ...
- list before: 1_0 <a_0, ..., a_i, ...>
- list after: 1_0 <a_0, ..., a_i, ...>

**store**

store picks two values from the data stack, i and a_i,(i being originally on top of the data stack). Then, it places value a_i at the index i in the list. (Compare it with ins operation below. store overwrites the existing value at the index i; ins inserts a new value at a given index.)

- stack before: i, a_i, ...
- stack after: ...
- list before: 1_0 <a_0, ...>
- list after: 1_0 <a_0, ..., a_i, ...>
length

Places the current length of the list on top of the data stack.

stack before: ...
stack after: n, ...
list before: l_0 <a_0, ..., a_n>
list after: l_0 <a_0, ..., a_n>

rmf

rmf removes the first element of the list.

stack before: ...
stack after: ...
list before: l_0 <a_0, a_1, ..., a_n>
list after: l_0 <a_1, ..., a_n>

rml

rml removes last element of the list.

stack before: ...
stack after: ...
list before: l_0 <a_0, ..., a_k, a_n>
list after: l_0 <a_0, ..., a_k>

rm

rm pops i from the data stack and then it removes i-th element of the list.

stack before: i, ...
stack after: ...
list before: l_0 <a_0, ..., a_i, ..., a_n>
list after: l_0 <a_0, ..., ..., a_n>

rmn

rmn pops n from the data stack and then removes n first elements from the list.

stack before: n, ...
stack after: ...
list before: l_0 <a_1, ..., a_n, ..., a_k>
list after: l_0 <..., a_k>
**ins**

*ins* pops two values from the data stack, \(i, v\) (\(i\) being originally on top of the data stack) and inserts the value \(v\) in the \(i\)-th position in the list. (Compare it with *store* operation. *store* overwrites the existing value at the index \(i\), *ins* inserts a new value at a given index.)

- **stack before:** \(i, a_i, \ldots\)
- **stack after:** \(\ldots\)
- **list before:** \(l_0 <a_0, \ldots, \ldots, a_n>\)
- **list after:** \(l_0 <a_0, \ldots, a_i, \ldots, a_n>\)

**car**

*car* removes the first element of the list and pushes it onto the data stack.

- **stack before:** \(\ldots\)
- **stack after:** \(a_0, \ldots\)
- **list before:** \(l_0 <a_0, \ldots, a_n>\)
- **list after:** \(l_0 <\ldots, a_n>\)

### 3.6.11 List stack

The list stack has a number of standard stack-based operations, that are equivalent to the appropriate operations from the data stack. All list stack mnemonics are named in the same way as their data stack equivalents with a leading “l”.

**lpop, lpopn**

*lpop* pops the top list from the l-stack. *lpopn* pops \(n\) elements from the l-stack.

**lswap**

*lswap* swaps the top two elements of the l-stack.

**ldup**

*ldup* duplicates the list on top of the l-stack.

**ldepth**

*ldepth* pushes the size of the l-stack onto the data stack.
lpop_p and lpush_p

lpop_p pops the list from the top of the l-stack and places it into the program list. The program counter and any other execution frame data-structures are not affected. lpush_p pushes the current program list into the l-stack. No execution data-structures are affected.

push_l

push_l pushes the current list, one by one, onto the stack. The first list element will be deep in the stack and the last list element will be on top of the stack.

3.6.12 quote, unquote

The quote operation, similarly to that in Lisp-like programming languages, marks the instructions following it as not-to-be interpreted by the execution engine. The instructions that are quoted become literals that do not have their usual meaning within the context of the execution, but become literal symbols only. Thus, quote quotes the block of code in the program list. This instruction sets the Boolean quoting flag to true. It will create a new empty list on top of the l-stack. When the quoting flag is true, the instructions from the program list are not executed, but they are appended to the list, on top of the l-stack.

unquote simply sets the quoting flag to false. This means that the next instruction after the unquote will simply be executed. The block of program between the quote and unquote will have been placed on top of the l-stack.

3.6.13 Arithmetic, logic and binary operations

The EVM does not differ substantially from any other standard register- or stack-based machine in terms of arithmetic, logic and binary operations. The EVM base machine provides all the basic arithmetic and logical operations.

There are a number of standard binary and logical operations implemented on the base level EVM machine: shl (shift left), shr (shift right), ushr, or, xor, and, not.

Similarly, the usual primitive arithmetic operations are implemented: add (addition), inc (increment), sub (subtraction), dec (decrement), mul (multiplication), div (division), rem (remainder/modulo), neg (arithmetic negation), pow (power). The reader should refer to the CD-ROM appendix for an exact description of all these operations.

3.6.14 Control flow

There are a number of standard control flow operations in the EVM base virtual machine that are inspired by other stack-based machines, most notably the Java Virtual Machine. Due to its character, the EVM also provides several unusual control operations. We will discuss these in more detail.
below. These instructions are introduced after extensive studies of automatic program generation techniques and are typically motivated by shortening certain frequently occurring control structures. Some of these are motivated by evolutionary computation literature (Schmidhuber, 1987; Olsson, 1995; Spector and Robinson, 2002), and some are derived from our own studies that are discussed in Chapter 9.

**halt**

**halt** halts the execution of the current execution frame unconditionally. In other words it terminates the current recursive call.

**halt0**

**halt0** represents a conditional **halt**. Same as **halt**, but only if the top element on the stack has value 0. Otherwise it does nothing.

**call**

**call** executes the instruction on top of the stack. An instruction number is taken from the top of the operand stack and it is executed on the current machine. If the instruction index is out of the current machine instruction range, it is automatically translated (normalised) into the appropriate range (by the modulo rule).

**for**

**for** executes a **for** loop. If the top element on the stack is zero, simply the next instruction will be executed. If the top element on the stack is not zero, then it decrements the top element on the stack and jumps to the beginning of the current program (to instruction 0 in the current program list). Effectively, the program will be executed from the beginning \(n\) times up to the for instruction (as long as the top element on the stack is not zero).

**exec**

**exec** pops the list from the l-stack and executes the program that this list represents in the current execution frame instance.

**jmp**

**jmp** pops the \(i\) value from the data stack and performs a relative jump into the position \(\text{ProgramCounter} + i\).
jmpsf

Jump Search Forwards. Pops an argument from the stack and searches forward for the first occurrence of this instruction in the program. If it is not found, it tries using the value modulo the instruction set size (i.e. the normalised instruction value). It executes the next instruction after the index is found. If it is not found, it does nothing. See also jmpsb.

jmpsb

Jump Search Backwards. Similarly to jmpsf, it pops an argument from the stack and searches backward for the first occurrence of this instruction in the program. If it is not found, it tries using the value modulo the instruction set size (i.e. the normalised instruction value). It executes the next instruction after the index found. If it is not found, this instruction does nothing.

ifeq

ifeq facilitates conditional execution. It pops the top element from the data stack and if it is 0, it will execute the next instruction in a sequence, skipping the following instruction. Otherwise, it will skip the next instruction in the sequence and execute the one after that.

ifgt

ifgt, similarly to ifeq facilitates conditional execution. It pops the top element from the data stack and if it is greater than 0, it will execute the next instruction in a sequence, skipping the following instruction. Otherwise, it will skip the next instruction in the sequence and execute the one after that.

goto

Implements an absolute jump operation. A new program index will be popped from the top of the stack and the program counter will be set to this new value.

3.7 Levels and Machines

The major feature distinguishing the EVM architecture from other traditional VMs is the hierarchical management of execution contexts and execution frames. Each operation in the EVM execution framework is represented by an integer in the program list. This integer is simply an index into the instruction list, which represents a given execution machine. For example, let us consider a machine $C$ (a calculator) capable of performing four arithmetic operations: add, sub, mul and div. Each of the operations have an assigned index in the machine list, for example: 0 (add), 1 (sub), 2 (mul), 3 (div). Consider a data stack $S$ containing the following data elements: 2, 3, 4, 5, where 2 is on the top of the stack and 5 at the bottom. Suppose we now have a program list
$P <0, 1, 2>$. $P$ executed on machine $C$ with a stack $S$ will represent the following arithmetic expression: $(2+3-4) \times 5$ and, upon execution of program $P$ on machine $C$ with the initial data stack $S$, the data stack will contain the single integer $5$ on the stack. Given a different machine $M$, the execution of program $P$ would result in a different outcome, depending on what operations are coded as indices $0, 1, 2$.

The machine sub-programs implemented under specific indices are called machine instructions. An instruction can be implemented as a single instruction, or sequence of instructions (a program) on a lower-level virtual machine. The chain of virtual machines ends at the Base Level, the Base Virtual Machine (BVM). Instructions of the BVM are called operations. The sequence of instructions or sequence of operations is called a program. All the operations discussed so far are implemented natively in Java and are handled by the BVM implementation. Any higher-level machine must refer to lower level virtual machines, or to the BVM itself. As explained, the hierarchy always terminates at the BVM level.

### 3.7.1 Execution frames

Execution frames are managed via a special execution frames list. This is similar to any other execution frame management system, for example the JVM (Venners, 1999). The difference is that instead of a usual method-call stack, or execution stack (such as in a JVM), we use a list data structure and programs have direct access to this list. There is a machine list attached to each of the execution frames. As explained earlier, a machine is a list of lists, where each individual list represents an implementation of a single instruction for the given machine. In other words, the machine is a list of lists of instructions, where each list implements a given machine instruction. If the given instruction is not one of the Base Machine operations, i.e. primitive operations for that machine, then the sequence must be executed on another lower-level machine. The Base Machine (at the lowest level) implements all the primitive operations that are not reified further in terms of more primitive units.

The execution frame manages a static list of machines, with two indices: one pointing to the current executing machine and one pointing to the higher-level virtual machine. The index pointing to the current machine is called $CM$ (Current Machine) and the index of the higher-level virtual machine is called $HM$ (Higher Machine). By default, $HM = CM + 1$, but this is not necessarily always the case. During runtime, when the program is executing, both indices can be manipulated independently. There are special instructions that manipulate both of the indices, which effectively switch the contexts of the machines. Within the execution frame there is one more important list, storing the chain of execution contexts. The contexts are created every time a subprogram has to be executed. Among other internal data the context contains:

- **program** – the program instruction list,
- **PC** – the program counter,
• the machine-related lists as described above:

  – **operand stack** – called also the stack or data stack,
  – **l-stack** – list stack, together with the top element, the **list**.

As mentioned earlier, EVM programs can potentially run indefinitely in the EVM abstract architecture. For experimental studies that would be undesirable. For our implementation, each thread of execution has a special limit to constrain the number of instructions each program in a multi-EVM environment can execute. Once the limit is reached, the program unconditionally halts. There is also an upper bound on the size of the data stack and l-stack data structures. There will be an unconditional program halt if any of the physical limitations is exceeded (that is, the computation requires more computing resources than are physically available for a given EVMI). Note that these limits are not imposed by the EVM abstract architecture, but only by the focus of the experimental work. Resource-constrained analysis of program space is a natural progression of our simplified implementation – any time the EVMA is implemented on existing computing hardware it is necessary first to analyse resource demands and constraints so that practical limits can be set.

The EVM offers powerful reflection and reification mechanisms via the operations described below. The computing model is relatively fixed at the lowest level, but it does provide the user with multiple computing architectures to choose from. The model allows the programs to reify the virtual machine on the lowest level. For example, programs can modify, add, and remove instructions from or to the lowest level virtual machine. Also, programs can construct higher-level machines and execute themselves on these newly created levels. In addition, a running program can switch the context of the machine to execute some commands on the lower-level or on the higher-level machine. All together, it provides high flexibility and capabilities for reifying individual EVM execution.

Each individual EVM cell can reference any other cell in the multi-EVM environment (the EVM Universe). This is achieved by using the first 32 bytes of the instruction to address any computer in the Internet, and the second 32 bytes for the index of the instruction on that machine. The following instructions are needed for manipulating and navigating execution frames and machines.

**mup, mdown**

*mup, machine up,* moves the current machine index to the next machine in the machine hierarchy. Respectively, *mdown, machine down* moves the machine index one down in the machine hierarchy. What this means is that the executing program can switch the actual machine that the instructions are executed on. If there are two completely different programs for two different machines, the programs can be concatenated with the appropriate *mup* and *mdown* instructions, assuming that the machines are in the appropriate hierarchy.

---

13For the EVMI used in this thesis, the scheme used the Internet Protocol (IP) addressing scheme.
hup, hdown

hup changes the index of the current high-level machine to the next one. If the index points to the current last machine in the chain and the last machine in the chain is not empty, a new empty virtual machine will be dynamically created and the new index will point to it. If the last machine is empty, the index will remain unchanged, pointing to it. hdown changes the index to the one lower. If the current HM index is 0, this instruction will do nothing.

Machine list management: mappend, mrm, mload, mstore

The program can freely manipulate instructions in the current machine. This is accomplished by a series of list-like operations that manipulate the machine list. mappend appends the current list to the end of the machine instruction list. mrm removes the last instruction of the machine instruction list.

mload pops an index $i$ from the data stack and reifies the $i$th instruction, i.e. it obtains the instruction sequence of the $i$th instruction of the current virtual machine and places it as a list on top of the l-stack. mstore pops the $i$ from the data stack and then pops the list $l$ from the l-stack. The list $l$ will then be inserted into the instruction set at the index $i$.

The machine manipulations expand the dynamic capabilities of the running EVM system, so it can create, change, refactor and re-arrange different subprograms and whole subsystems that can be characterised by completely new “languages”. A given sequence of integers can mean semantically different things on two different virtual machines. However, on the other hand, the same structural patterns (the same programs) can be re-used in different contexts and different virtual machine settings. The architecture is geared towards exploiting any regularity that may occur in the computational space: structural patterns, behavioural patterns and mixtures of both of these. In this respect the EVM architecture is a unique computational model that bridges different paradigms to deliver a flexible and expressive computational model suitable for automated program generation.

3.7.2 Context switching: eup, edown

The EVM architecture provides an expandable computational model, where all the computational elements can be reified, modified or re-implemented during runtime. In technical terms, all the instructions of a given virtual machine can be modified or re-implemented by the running program itself. The program can also modify or re-implement control flow instructions. In some situations, the executing program must be able to access the meta-data-structures of its own execution frame, or the execution frame of its parent, or of one of its children.

To enable programs to freely access all those different contexts, we have implemented a verbose mechanism for context switching. It works in a similar way to higher-order machine switching. There is a special index, called execution Index, or eIndex, that contains the index of the current execution
context pointing to the Execution Frame List, *EFL*. *eIndex* is thus a pointer to the current execution context from existing Execution Frames currently on the execution list.

There are two operations that manipulate the *eIndex*: `eup` and `edown`, that respectively increment and decrement the value of *eIndex*.

![Diagram](attachment:image.png)

**Figure 3.5**: EVM execution frames and dynamic context switching.

### 3.7.3 Reification and reflection limits

The *eIndex* value that is associated with a given Execution Frame is one of the data structures that is not accessible to programs from other execution frames. For example, a program can manipulate the Program Counter of another execution frame, or implement any other arithmetic or stack, l-stack manipulation routines, but there is no mechanism to re-implement the reflection and context switching mechanisms. The current context switching, virtual machine hierarchy and higher-order machine manipulation implementations are outside of the reflective tower and cannot be re-implemented by the EVM program itself. Any other computation can be reified and re-implemented by the EVM program alone. It was our design decision to terminate the reification capabilities at that level. From our earlier studies and experiments it seemed unlikely that this mechanism will need to be reified or re-implemented dynamically for any specific problem domain. Further studies however may indicate areas where such reification capabilities could potentially be beneficial.

### 3.7.4 Instruction probability manipulation

This EVMI prototype has been expanded by adapting bias-optimal search primitives (Levin, 1973). This is to enable studies on bias-optimal search techniques in comparison with random search and various genetic algorithm models.
For the implementation, we have used incremental search methods (Schmidhuber, 2004). The idea is relatively simple: there is a built-in probability distribution mechanism that stores relative probabilities of each of the instructions of a given virtual machine, so, automated stochastic search procedures can use this information in the search process. The probability manipulation is controlled by the program alone. The probability distribution is organised into a data structure that we call a page. A single page contains a probability distribution for all instructions. Each such page can be stored on a special probability distribution stack. This allows managing different probability distributions for different contexts and reverting back to the previously established probability distribution if the new, changed, one is not performing as well.

One can combine several methods together. For example, it is possible to construct a generator of problem solver generators and employ multiple meta-learning strategies as described in more detail by Nowostawski, Purvis, and Cranefield (2004). This is an area for further explorations.

**qboost**

qboost boosts (increments) the probability of all the instructions in the program by a pre-specified value.

**qinc**

qinc increments the probability of all the instructions from the program block between the previous qinc or qdec and this current qinc instruction.

**qdec**

qdec decrements the probability of all the instructions from the program block between the previous qinc or qdec and this current qinc instruction.

**qpush, qpop**

qpush ushers the current probability distribution model onto the probability distributions stack, q-stack. qpop pops the probability distribution model from q-stack and instantiates it as a current model.

### 3.7.5 Example EVM assembly programs

Below we present some examples of EVM assembly code for various tasks. This is to provide some code snippets demonstrating some of the programming features offered by the EVM assembly operations. Also, we compare human generated code and some of the more interesting automatically generated (discovered) constructs through various learning mechanisms.
• The first task consists of checking if a given integer is odd or even. The initial integer number is placed on the top of the operand stack. The result 1 should be placed on top of the stack if the input number is odd, and 0 if the number is even.

Human generated solution.

```plaintext
const_2  /* place 2 on top of the stack */
swap     /* swap the arguments on the stack */
rem      /* perform the number module 2 operation */
```

Solution found by a genetic algorithm.

```plaintext
const_2  /* place $2 on top of the stack */
and      /* perform logical AND bitwise operation */
```

• Self-replicating program, discovered through genetic algorithm search.

```plaintext
lpush_p  /* copy program into the list */
push_1   /* copy list onto the stack */
plength  /* copy the program length onto the stack */
roll     /* revert the order of instructions */
        /* so that the first program instruction */
        /* is on top of the stack */
```

• Calculate a factorial of an input integer on top of the stack. This program uses a plain relative jmp instruction (it does not use the exec or for).

```plaintext
dup      /* the tmp factorial */
dec      /* decrement the counter */
dup      /* duplicate the counter */
const_1   /* place $1 */
sub       /* decrement the top */
const_m1  /* place $-1 */
push 10   /* place $10 */
mul       /* -1 * 10 on top of the stack */
swap      /* check if the counter is 0 */
ifneq     /* if counter > 0, */
mul       /* multiply to get the tmp factorial */
        /* value and jump to program start */
const_m2  /* otherwise jump out of the program */
jmp
```

The program above is quite complex and difficult to be found by any of the automated search mechanisms we have used. The same factorial semantics can be achieved by a shorter program with the use of the exec routine:
lpush_p
dup
dec
dup
ifgt
exec
halt0
mul

- Example of higher machine manipulation. This code snippet appends a new instruction into the higher-level machine at position 2. This new instruction basically calculates addition. (The instructions indexed 0 and 1 are initialised as mup, and mdown respectively).

```plaintext
quote
add /* index of 'add' instruction on the stack */
unquote
hup /* change the HM = HM + 1 */
mappend /* add new instruction */
mup /* CM = CM + 1 */
2 /* execute new instruction indexed 2 */
/* on the higher level machine */
```

3.8 Static and dynamic aspects of the EVM cell

The initial values of the data structures, the state of the execution, the virtual machine chain (the hierarchy) and the actual program to be run within this environment, can all be represented statically as a single sequence of BVM operations, executed on the BVM. Therefore, any given runtime state of EVM computation, on multiple virtual machines, with multiple contexts, can be ultimately mapped to a sequence of instructions of the BVM. However, analysis of computation on the BVM level and, more importantly, automated program generation on the BVM level alone is not the best approach for all sets of problems. The EVM architecture is designed so it can benefit from a dynamic management of various computational data structures and behaviour, that can be manipulated, created and disposed dynamically during the runtime.

During the course of execution, an initial program can create, use and dispose of an arbitrary number of intermediate virtual machines running programs in appropriate languages. The trace of such an execution can be investigated and analysed, leading to a better automatic and autonomic generation of suitable virtual machines for given tasks. The implemented EVM architecture helps to model and keep track explicitly of dynamical properties of a running program, which otherwise may be completely implicit and often, intractable.

Many cells may be involved in a single thread of computation and it may become complicated to move data between all of them. A fast parallel implementation (in hardware) is difficult to design,
but not impossible. We envision, in the future, special self-adapting operating systems executing on re-configurable hardware implementations, such as an FPGA-based TTA (Transfer Triggered Architecture) for example (Corporaal, 1997). We will discuss possible directions of research in Chapter 10.

Note, that the actual management of programs within the entire EVM environment (or EVM Universe) is not part of the basic BVM implementation. These aspects are part of a given experimental setup used in a particular context. Such setups are explored and discussed in Chapter 9.
3.9 Efficient EVM implementation

Due to the computational complexity of EVM program execution, a substantial amount of time has been dedicated to investigations of efficient BVM implementations. The initial Java-based EVM environment proved valuable for fast prototyping and testing various research ideas related to EVMA/EVMI internals and organisation. However, the Java implementation turned out to be prohibitively slow for larger problems and non-trivial task sets. The need for an optimised and faster EVM implementation turned out to be essential for conducting more advanced and complex EVM research and development.

In this section we present, discuss, and review our efforts concerned with the second generation EVM implementation. We will compare the results obtained to the initial preliminary EVM implementation done in Java and provide directions for further improvements.

To facilitate faster EVM runtime capabilities and optimised implementation, we have used a new 3-layered architecture. The user interface (Figure 3.4) and GUI-based presentation layer are re-used from the first generation Java-based EVM IDE environment. This provides a range of capabilities ranging from preparing and configuring an experimental setup, tuning the parameters and observing on-the-fly the evolution of the running EVM system. The actual EVM kernel (EVMK) is re-implemented and interfaced with the Java-based front end. The EVMK is divided further into two parts: machine-independent code, written in C and architecture-dependent code written natively in assembly. The division between the user-driven frontend (written in Java) and the EVM kernel back-end is straightforward from the software engineering point of view. Much of the existing Java-based code is reused for that purpose. However, the choice of components rewritten natively in assembly for the efficient EVM kernel implementation needed further considerations. Native utilisation of a given computing architecture (written in assembly) usually results in smaller and faster code (Dandamudi, 1998). However there is a trade-off between the productivity and maintainability of assembly code and the portability and productivity of a C-based implementation. On one hand writing portable C code is preferable to hand-crafting assembly code. However, lower productivity and higher development costs for assembly code are justified as long as substantial gains in performance can be achieved. To establish the potential benefits of reimplementing parts of the EVM kernel natively in assembly code, a series of tests and performance analysis have been conducted. We have tested various implementations of the basic EVM on register-based commodity hardware tuned for best possible performance.

There are three main EVMK considerations:

1. stack and stack operation implementation
2. list and list operation implementation
3. primitive operation implementation (arithmetic, logic, control operations, etc.)
3.10 Testing environments

Our EVM implementation has been tested on a variety of contemporary register-based systems. We will refer to them as Intel Core2, Pentium4, PentiumD, AMD Opteron2 and Sun Sparc T2000 system with Niagara-T1 CPU. We have used several different Java runtime versions and we have observed substantial performance gains when using JDK 1.6, therefore, all results below have been collected on Java SE 1.6.0_01-b06.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Core2</th>
<th>Pentium4</th>
<th>PentiumD</th>
<th>Opteron2</th>
<th>T2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Core2 Duo</td>
<td>Pentium 4</td>
<td>Pentium D</td>
<td>Opteron 2212</td>
<td>Niagara T1</td>
</tr>
<tr>
<td>Cores</td>
<td>2</td>
<td>2 (HT)</td>
<td>2</td>
<td>4</td>
<td>32 (8x4)</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.0GHz</td>
<td>3.6GHz</td>
<td>2.8GHz</td>
<td>2.0GHz</td>
<td>1GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>2x32kB</td>
<td>12kB+16kB</td>
<td>24kB+32kB</td>
<td>2x128kB</td>
<td>16kB+8kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4MB</td>
<td>2x1MB</td>
<td>2x1MB</td>
<td>4x1MB</td>
<td>3MB</td>
</tr>
<tr>
<td>FSB [MHz]</td>
<td>667</td>
<td>800</td>
<td>533</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>RAM</td>
<td>2GB@677</td>
<td>1GB@533</td>
<td>1GB@533</td>
<td>2GB@677</td>
<td>16GB@533</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.6.21.3</td>
<td>2.6.17</td>
<td>2.6.21.1</td>
<td>snv.65</td>
<td>snv.57</td>
</tr>
<tr>
<td>gcc</td>
<td>4.1.2</td>
<td>4.1.1</td>
<td>4.1.2</td>
<td>3.4.3</td>
<td>3.4.3</td>
</tr>
</tbody>
</table>

Table 3.1: Testing environments.

Note: Front-Side-Bus (FSB) is a term describing a Processor (CPU)-to-system memory data bus. gcc is the compiler of the GNU environment.

3.11 EVM stack

3.11.1 Stack as a data structure

Any stack implementation must offer at least four operations:

1. **init** – to initialise a stack in memory
2. **pop** – to pop an integer from the top of the stack
3. **push** – to push an integer onto the top of the stack
4. **free** – to release memory and other resources utilised by the stack

The EVM is a stack-based virtual machine. This means that each primitive operation of the EVM takes and deposits its arguments and results from the operand stack. The EVM operand stack is the most heavily used data structure within the EVM architecture. Almost every processor operation needs to access at least once the operand stack: to fetch the operands from and to put the results of
the operations back onto the stack. The stack is at the heart of any EVM machine. Register-based
machines also utilise the stack. They usually map a certain region of memory to be dedicated as
stack storage. Because arbitrarily long stacks cannot be fitted into the internal memory of the CPU
itself (the registers and cache) stack operations are usually quite slow compared to the register-based
operation of commodity CPUs. A dedicated hardware implementation of the stack would make the
best EVM implementation. However, on the existing commodity CPUs (Intel, AMD, Sparc) the stack
is implemented through general memory mapping and the CPUs do not provide hardware-support for
stack operations. In other words, stacks on register-based machines must be implemented in software.

3.11.2 Stack implementation in software

To evaluate the performance of the EVM implementation, several scenarios of stack implementation
have been devised, tuned and tested. Two implementations in Java, two in C and three in assembly
language have been proposed and evaluated.

Appropriate implementations will be shown as code listings in Java, C and AT&T/Unix assembly
language syntax listings for the Intel IA-32 architecture. Both C and Java implementations are archi-
tecture independent. This means that both Java and C implementations do not use or refer directly
to any particular architecture or CPU-dependent features. The mapping of the high-level implement-
tation (both in Java and C) down to a particular computing architecture is achieved (automatically)
by means of automated compilers, that have the responsibility of generating efficient and optimised
code. In the context of C this is done by GCC\textsuperscript{14} and optimisation flags. In Java this is done through
the just-in-time compiler (JIT). Assembly language implementations inherently use a particular CPU
instruction set directly, and automated optimisation is achieved by crafting individual assembly in-
structions manually by the programmer. This highly reduces the programmer productivity but may
potentially result in highly optimised and efficient code not achievable by automated code generators
such as gcc compiler.

3.11.3 Stack tests

To test the performance of a particular stack implementation, a simple scenario of repeated operations
(push and pop) has been run on a stack up to 5000 items in size. The timed scenario consists of
repeating 500,000 times the following pattern: a stack is initialised; then 5000 consecutive \texttt{int} values
are pushed onto the stack; then all the 5000 values are \texttt{pop}ped one by one from the stack. In other
words, we have timed how long it will take to fill up and remove all elements from a 5000 long
stack 500,000 times. The whole experiment has been repeated at least 11 times on each hardware
configuration to establish an average time. This is what has been timed and is reported later in the
comparative performance listing in Table 3.2.

\textsuperscript{14}http://gcc.gnu.org/
### A: Java implementation with `java.util.Stack`

Java provides a stack implementation through one of the Collection types. The first implementation thus utilises the Stack implementation shipped with the Java Runtime Environment (JRE). The current Java virtual machine provides boxing and autoboxing of a primitive `int` types, however, this feature has not been used in our implementation. Instead, we have directly converted primitive `int` into an `Integer` object. The `java.util.Stack` class already implements the initialisation, pop and push operations on the stack. The memory cleanup is done by the garbage collector automatically.

```java
import java.util.Stack;

...

public static void main(String[] args) {
    int LOOP = 500000;
    int LENGTH = 5000;

    // main loop
    for (int j = 0; j < LOOP; j++) {
        // create an instance
        Stack s = new Stack();
        // initialise all the elements
        // by pushing consecutive integers onto the stack
        for (int i = 0; i < LENGTH; i++) {
            s.push(new Integer(i));
        }
        // pop each element from the stack and
        // test if no errors on the stack
        for (int i = 0; i < LENGTH; i++) {
            int e = ((Number)s.pop()).intValue();
            if (e != LENGTH - 1 - i)
                System.out.println("[ERROR] stack corrupted");
        }
        s = null;
    }
}
```

### B: Custom stack in Java

As one may expect, the built-in object-oriented `java.util.Stack` implementation is not particularly suited or efficient for manipulating a large number of primitive integers. We have implemented a simple stack data structure backed up by an array of primitive integers. Java handles primitive types efficiently, as they map naturally to the underlying computing architecture and operations on primitive
types are usually much more efficient than object-oriented data types. The results of the performance
testing below confirmed that the custom stack implementation in Java is much more efficient than
the object-oriented stack. In particular, manual array manipulations tend to be more efficient than
the use of Java collections. It is worth noting that performance is traded against readability and
maintainability of the program source.

```java
public final class IntStack {
    // data store, storing the stack data
    private int[] data;

    // current size of data on the stack
    private int size;

    // Initialise the memory space for the stack
    public IntStack() {
        this.data = new int[Constants.INT_STACK_SIZE_INITIAL];
        this.size = 0;
    }

    ... // pops an element from the stack
    public int pop() {
        if (this.size == 0) return 0;
        return this.data[--this.size];
    }

    // pushes an element onto the stack
    // expands the memory storage if necessary
    public IntStack push(final int value)
        throws SizeLimitExcededException {
        if (this.size > Constants.INT_STACK_SIZE_LIMIT) {
            throw new SizeLimitExcededException();
        }
        if (this.size == this.data.length) expandDataBuffer();
        this.data[this.size++] = value;
        return this;
    }

    ... // expand the data buffer
    private void expandDataBuffer(){
        final int[] old = this.data;
        final int[] tmp = new int[this.data.length * 2];
        System.arraycopy(old, 0, tmp, 0, this.data.length);
    }
}```
3.11.6 C: C1, Initial C implementation

The C language offers programmers the flexibility of a higher level programming language together with low-level features that are close enough to the underlying machine to achieve excellent performance. We have decided to use the C language as the basis for the EVMK implementation with some of the mission-critical functions written natively in assembly language. Despite its age, C continues to offer one of the best trade-offs between (native) fast execution, portability, and relatively high productivity. Our first C implementation is close to the object-oriented stack implementation of approach B discussed above. The stack is represented as a `struct` with its first two integers holding the current indices of the current (top) and the maximum stack size, followed by the array of integers representing the actual stack. The top element can be pushed and popped from the stack by using the index (`top`) of the `bottom_p` array.

```c
// Main integer stack data structure
typedef struct {
    int top;
    int max;
    int bottom_p[];
} stack_t;

// Initialise the stack
void
stack_init(stack_t **s, int stack_max_length)
{
    *s = malloc(sizeof(stack_t) + 4 * stack_max_length);
    (*s)->top = 0;
    (*s)->max = stack_max_length;
    if (s == NULL) ERROR_STACK = -1;
}

// Release the memory used by the stack
void
stack_free(stack_t *s)
{
    free(s);
}
```
// Push an element on top of the stack
// memory is pre-allocated – check the limit

void push (stack_t *s, const int value) {
    if (s->top == s->max) {
        ERROR_STACK = ERROR_STACK_FULL;
    }
    s->bottom_p[s->top++] = value;
}

// Pops an element from the stack

int pop (stack_t *s) {
    if (s->top == 0) {
        ERROR_STACK = ERROR_STACK_EMPTY;
        return 0;
    }
    return (s->bottom_p[--s->top]);
}

Note, that all the C implementations were compiled with the gcc compilers specified in Table 3.1 together with the −O3 optimisation flag turned on. On all of the Intel-based machines, the compiler generated exactly the same machine code. The machine code generated for AMD and Sparc platforms differed due to the nature of the ISA (Instruction Set Architecture) which makes direct performance comparison impossible.

3.11.7 D: C2, C implementation without indexing

Reimplementing the stack data structure in such a way as to not keep the index but the address of the top element of the stack, may improve the performance. Instead of storing the base address and the offset, the memory location is stored directly. This, in theory, may result in slightly faster execution times. This has been implemented as our second, modified C implementation.

// Main integer stack datastructure
typedef struct {
    int* current;
    int* top;
    int* bottom;
} stack_t;

...
As can be seen later on in the performance results, this indeed has improved the overall performance of the stack implementation.
3.11.8  E: C3, C implementation, final refinement

For the final refinement of the C implementation the structure of if statements has been changed. This, one may think, insignificant change results in a slightly different arrangement of the code in the memory and, more importantly, in the way conditional execution is conducted. Most CISC (Complex Instruction Set Computers) architectures, such as Intel and AMD family always prefetch the positive branch\(^\text{15}\) of the conditional instruction into the CPU pipeline. When the negative branch is going to be executed, the prefetched positive branch needs to be disposed and the negative branch needs to be fetched instead. This means, if a given conditional statement takes the negative branch most of the time, there will be a noticeable performance penalty when compared to conditional statement which takes the positive branch most of the time. In other words, it pays to arrange your code so as to take the positive branch most of the time. This is how the code has been changed as compared to C2. The stack and all related data structures remained the same - the only change is in the actual code for push and pop functions.

```c
void push (stack_t *s, int value) {
    if (s->current != s->top)
        *(s->current++) = value;
    else
        ERROR_STACK = ERROR_STACK_FULL;
}

int pop (stack_t *s) {
    if (s->current != s->bottom)
        {
            s->current--;
            return *s->current;
        }
    else
        {
            ERROR_STACK = ERROR_STACK_EMPTY;
            return 0;
        }
}
```

\(^{15}\)The positive branch is the one that is executed if the condition in the conditional expression is positive, or equivalent to a logical positive truth value, usually expressed as the Boolean value 1 or true. The negative branch on the other hand is the one that is executed if the condition is negative or false.
3.11.9  F: Asm 1, base assembly implementation

The compiled optimised C code can be further improved through hand crafting stack routines directly in assembly language. This results in certain performance gain, since a smart programmer can outsmart a smart automated compiler\(^{16}\). This is mainly due to a lower number of instructions and fewer clock cycles needed to execute the code, and smart L1/L2 cache utilisation. The base assembly language implementation has been done for the Intel x86 architecture and tested on various Intel environments. The initial implementation code is provided below. It uses memory as “a storage” backing up the stack data structure. The memory used for the EVM stack is allocated on the CPU stack.

\^\text{16}\text{This is not as easy at it may seem. Modern compilers and code optimisers are increasingly complex systems that produce efficient machine code.}

```c
/* Global macro for convenience */
#define BEGIN(x) \
    .globl x:\
    .align 8:
    .type x, @function:\

x:

#define END(x) .size x, . - x

...

/* Stack initialisation */
/* Place the start of stack memory into %esp */

/* We use the actual CPU stack register */
/* for storing our internal data stack */
BEGIN(init_stack)
    subl STACK_START, %esp
    movl $0, %eax
    pushl %eax
    addl $4, %esp
    addl STACK_START, %esp
    ret
END(init_stack)

/* remember that CPU stack based on the %esp */
/* register grows downwards in memory */
BEGIN(push)
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    popl %ebp
    subl STACK_START, %esp

```
The assembly code follows the AT&T/Unix syntax. The preamble and finishing instructions for the function definitions have been shortened through the BEGIN and END macros.

### 3.11.10 G: Asm 2, MMX-based assembly implementation

Since the Pentium III, 8 extra 64-bit registers have been introduced into the generic Intel x86 architecture. These are so-called multi-media extension registers that allow faster and more robust manipulation of multimedia data. These registers can store data in the form of long integers or floating point numbers. MMX registers are to be used with SIMD-type (single instruction multiple data) instructions. These instructions allow a single CPU instruction to operate on packed data structures of 2 double precision or 4 single precision floating point numbers, or 2, 4 or 8 double words, words and bytes, respectively. We used these registers to store stack meta-data (size, limit, current top item address) together with 3, 4 and finally 5 top elements of the stack directly in the registers. This in turn will mean, that operations that need to fetch from or write to the top of the stack will operate on the registers directly.

```assembly
BEGIN(x) \
.globl x: \
.align 8: \
.type x, @function; \
x:
```

85
# define END(x)  .size x, . - x

...  

/* prepare the memory through call to system malloc function */
/* and zero the value of MMX_D register, which will be used */
/* to store the current stack size. */
BEGIN(stack_init)
    pushl %ebp  /* store the execution frame */
    movl %esp, %ebp  /* and store frame info */
    movl 8(%ebp), %eax  /* get the argument */
    pushl %eax  /* malloc argument onto stack */
    call malloc  /* allocate memory */
    movd %eax, %mm1  /* store the evm stack address */
    pxor %mm0, %mm0  /* zero the counter */
    leave
    ret
END(stack_init)

BEGIN(stack_free)
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    pushl %eax
    call free
    pxor %mm1, %mm1
    leave
    ret
END(stack_free)

/* mm0 stores the current stack size */
/* mm1 stores the current stack start address */
BEGIN(pop)
    movd %mm0, %eax
    dec %eax
    movd %eax, %mm0
    movd %mm1, %ecx
    subl $4, %ecx
    movd %ecx, %mm1
    movd %mm7, %eax
    movq %mm6, %mm7
    movq %mm5, %mm6
    movd (%ecx), %mm5
    ret
END(pop)
3.12 Performance results

All test cases from A to G have been tested on all available runtime environments, with the exception of the assembly code. We have only implemented the assembly tests for the Intel x86 architecture. The AMD assembly language differs slightly due to its 64-bit registers and stack alignment (it follows the modified x86_64 architecture). Sparc assembly language differs substantially – there are completely different sets of registers and the internal CPU architecture is generally different (RISC\(^{17}\) vs. CISC\(^{18}\)). No assembly language implementation has been provided for testing purposes there.

The stack tests were conducted only for estimation purposes, to establish the best overall design of the EVMK. The EVMK has been implemented as a pure C implementation that can be compiled on any platform (platform independent C code). This is augmented with assembly code implementation for certain parts written directly in assembly for specific commodity architectures (Intel x86 and AMD x86_64). Thus the EVMK can be compiled with the optional architecture-specific assembly modules. In the near future we plan to re-implement EVMK to run natively on commodity graphics processors (i.e. on a GPU, a graphics processing unit), as this promises substantial performance benefits (Luebke, Harris, Krüger, Purcell, Govindaraju, Buck, Woolley, and Lefohn, 2004).

There are many different factors that influence a given program execution. The majority of performance enhancements is done by the programmer at the higher level through the structure of the algorithm itself. In the case of a stack implementation there is not much a programmer can actually do. The operations are relatively simple, and there are not many possible implementations. The bene-

\(^{17}\)Reduced Instruction Set Computers.
\(^{18}\)A RISC architecture has a completely different organisation of registers and a so-called Register Window, which is used to pass input-output parameters of function calls. In CISC this is again architecture dependent, but most architectures pass input parameters on the CPU stack and output results through registers.
fits boil down to efficient use of CPU instructions and efficient utilisation of the internal CPU registers and cache (L1 in particular). Java does not give the programmer enough control to take advantage of the underlying platform. The C and assembly language implementations try to take the full advantage of the underlying hardware.

The total time in seconds of performing the stack push/pop test (in total, 5,000 * 2 * 500,000,000 stack operations) is presented in Table 3.2. The time is in seconds, ± twice the standard deviation.

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Pentium 4</th>
<th>Pentium D</th>
<th>Opteron2</th>
<th>T2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Java 1</td>
<td>153 ± 0.5</td>
<td>285 ± 0.4</td>
<td>334 ± 2.4</td>
<td>186 ± 0.2</td>
<td>1518 ± 28</td>
</tr>
<tr>
<td>B: Java 2</td>
<td>49.06 ± 0.9</td>
<td>51.16 ± 0.6</td>
<td>55.18 ± 2.5</td>
<td>63.82 ± 0.4</td>
<td>459 ± 21</td>
</tr>
<tr>
<td>C: C 1</td>
<td>28.07 ± 0.04</td>
<td>28.23 ± 0.2</td>
<td>32.02 ± 0.05</td>
<td>38.9 ± 0.01</td>
<td>249 ± 0.05</td>
</tr>
<tr>
<td>D: C 2</td>
<td>29.67 ± 0.04</td>
<td>25.44 ± 0.6</td>
<td>28.77 ± 0.02</td>
<td>37.6 ± 0.02</td>
<td>227 ± 1.3</td>
</tr>
<tr>
<td>D: C 3</td>
<td>24.39 ± 0.1</td>
<td>24.06 ± 0.2</td>
<td>27.22 ± 0.03</td>
<td>34.2 ± 0.07</td>
<td>231 ± 0.5</td>
</tr>
<tr>
<td>E: Asm 1</td>
<td>21.51 ± 0.04</td>
<td>19.18 ± 0.28</td>
<td>22.07 ± 0.17</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>F: Asm 2</td>
<td>19.42 ± 0.3</td>
<td>20.79 ± 0.15</td>
<td>23.26 ± 0.08</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.2: Performance results: time summary.

In Table 3.3 we have normalised the columns with respect to the last row (with the exception of Opteron2 and T2000 which were normalised with respect to the C3 implementation). This gives an overview of the speedup gains as compared to the MMX-based assembly implementation. The tests show, within a single column, that the efficient assembly implementations are the fastest, giving roughly around a 2.4 and 1.4 speedup factor when compared to Java and C implementations respectively.

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Pentium 4</th>
<th>Pentium D</th>
<th>Opteron2</th>
<th>T2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Java 1</td>
<td>7.87</td>
<td>13.70</td>
<td>14.35</td>
<td>5.44</td>
<td>6.58</td>
</tr>
<tr>
<td>B: Java 2</td>
<td>2.52</td>
<td>2.46</td>
<td>2.37</td>
<td>1.87</td>
<td>1.99</td>
</tr>
<tr>
<td>C: C 1</td>
<td>1.45</td>
<td>1.35</td>
<td>1.37</td>
<td>1.14</td>
<td>1.10</td>
</tr>
<tr>
<td>D: C 2</td>
<td>1.52</td>
<td>1.22</td>
<td>1.23</td>
<td>1.10</td>
<td>0.99</td>
</tr>
<tr>
<td>E: C 3</td>
<td>1.25</td>
<td>1.15</td>
<td>1.17</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>F: Asm 1</td>
<td>1.10</td>
<td>0.92</td>
<td>0.94</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>G: Asm 2</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.3: Performance results: implementation speedup.

It is not viable to compare the implementations and execution times across various machines, but, we were tempted to do so nevertheless. Note, these results are for a simple benchmark and should be treated as rough estimates only. In Table 3.4 all the rows have been normalised to the first column. Generally, the Intel Core2 Duo performs well compared to other platforms, especially for large and complex problems (such as Java implementation A). In most cases it beats or matches
the Intel Pentium 4 with almost double the clock speed and double the power consumption. The processing per Watt has been clearly hugely improved in the recent line of Intel CPUs. On the other extreme, the Sun Sparc T1 CPU seems to be largely disadvantaged. It is important to note however, that T1 has overall 32 hardware strands that can be run concurrently, and our tests were only utilising a single strand for the stack tests. Parallel and throughput processing testing, where T1 strands can be pipelined for increased performance, provide results close to those of the Intel Core2 CPU. The computing power to Watt ratio for the T1 CPU is currently the best of the existing CPU designs.

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Pentium 4</th>
<th>Pentium D</th>
<th>Opteron2</th>
<th>T2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>1.00</td>
<td>1.86</td>
<td>2.18</td>
<td>1.22</td>
<td>9.92</td>
</tr>
<tr>
<td>B:</td>
<td>1.00</td>
<td>1.04</td>
<td>1.12</td>
<td>1.30</td>
<td>9.36</td>
</tr>
<tr>
<td>C:</td>
<td>1.00</td>
<td>1.00</td>
<td>1.12</td>
<td>1.36</td>
<td>8.72</td>
</tr>
<tr>
<td>D:</td>
<td>1.00</td>
<td>0.85</td>
<td>0.96</td>
<td>1.26</td>
<td>7.65</td>
</tr>
<tr>
<td>E:</td>
<td>1.00</td>
<td>0.98</td>
<td>1.11</td>
<td>1.40</td>
<td>9.47</td>
</tr>
<tr>
<td>F:</td>
<td>1.00</td>
<td>0.89</td>
<td>1.02</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>G:</td>
<td>1.00</td>
<td>1.07</td>
<td>1.19</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.4: Performance results: speedup between CPU architectures.

### 3.13 Conclusion

In this chapter we have presented the details of the EVMI implementation. We discussed the major differences between the EVMI and the EVMA, and provided details of the internal organisation, instruction set and general computing paradigm of the EVMI. In the next chapter we will provide more theoretical reflections on EVMI and provide some formalisms in the context of the general theory of computation.
Chapter 4

Theory of Computation
4.1 Overview

The objective for this chapter is to present the EVMA and EVMI from an abstract computational perspective and motivate our interests and objectives. We focus on the EVMA as a generalised computational model. We present and discuss our architecture as a model of parallel asynchronously interacting computation. We will present the overall landscape of contemporary computational theories with respect to asynchronous massively parallel systems and discuss where the EVM model fits in.

We start our discussions with the basic concepts such as algorithmic computation, algorithms, Turing machines, and other related concepts that are necessary for establishing the scope and contribution of the EVMA from a computational perspective. We discuss the traditional models of computation, such as Turing machines, and then discuss concepts related to extended notions of computation. We introduce the notion of supertasks and hypercomputation. After that, we come back to the basic notions of computation, and we propose a single, unified view that we refer here as the reasonable model of computation. We explore the notion of interactive computing and we discuss how the novel representation provided by the EVM model differs from the sequential computational models of traditional computing frameworks. We argue that to make progress in the area of parallel processing, especially within multi-core and multi-chip computing architectures, a paradigm shift is necessary: from monolithic designs to distributed, loosely coupled and ad hoc architectures. EVMA augments and generalises some of the existing process-oriented computational models to the notion of computation with large numbers of asynchronously communicating processing units.

4.2 Motivation

The main motivation for EVMA is to provide a new massively parallel and asynchronous computational architecture that can be programmed easily in a fully automated way. Human programmers have difficulty dealing with concurrency, and programming massively parallel and asynchronous systems is a challenging or in some cases unachievable task. We envision that in the future programming massively parallel asynchronous systems will be done exclusively by the computers themselves\(^1\). In the EVMA the basic means of programming will not be hand-crafting of algorithms by human programmers – human programmers will only provide objectives and resources. The actual programming

---

\(^1\)Programming future systems will be similar to modern CPU and computer designs that are almost exclusively developed with the use of computer-aided design tools or in fully automated ways by computers themselves.
and problem solving will be done by automated program generators instead².

Traditional human-centred computational frameworks and computing languages have not been driven by the goal of fully autonomous and automated programming. Existing computational models and architectures are exclusively designed for human operators, in such a way that it makes it easier for humans to manage the complexities of the software executed on a given architecture. This paradigm is partially driven by historical reasons and partially because of the underlying theory of computation itself. On one hand, early computing systems were small, so well planned and well designed monolithic sequential software systems were abundant and more than appropriate. Systems were planned to fulfil a certain single task and they were largely based on simple sequential designs. Nowadays two important phenomena can be observed:

First, there is a rapid growth in the complexity of software projects. This, together with a fast evolution of specifications and changes of requirements make large monolithic designs inappropriate. In the past, large systems were designed and implemented over relatively long periods of time, and they worked almost unchanged for a considerable time. Nowadays, systems must be rapidly redesigned and re-implemented to fit particular needs that change quickly. There is no place anymore for long design phases – instead, software is designed, deployed and evolved continuously. This is visible in most of the contemporary software development corporations such as Google (Google Inc., 2007).

Second, there is a visible paradigm shift in the way software is designed and deployed. This is driven by the change in hardware architectures: from fast single CPU mainframes, to slower (in clock speed) but integrated and highly parallel multi-chip and multi-core based cluster computing facilities. Again, looking at the forefront, for example at Google, it becomes apparent that the design and deployment model is changing. From single static designs and isolated deployments on a client machine, new centrally managed, cross-linked and rapidly or even continuously changing deployments on large cluster-based distributed computing farms are becoming more abundant. To address these changing needs a paradigm shift is needed, changing the way we think of computation. On one hand, there is a need for more flexible, parallel and asynchronously communicating system processor architectures. On the other hand, there is an interest in adapting bio-inspired computing models that could help us manage, design and deploy such systems. Note that bio-inspired models can also (in certain circumstances) facilitate rapid evolution of software systems.

Thus there is a motivation to provide a generalised computational modelling framework. The physical world is best modelled as a collection of a large number of autonomously and asynchronously interacting entities. In computational terms, the physical world exhibits a high degree of concurrency. We have tried to design EVMA such that real-world phenomena could be modelled in computational terms through the utilisation of massive parallelism and asynchronous interactions.

²Of course, the current EVMA research is not trying to tackle the actual problems of parallel programming – it is rather about providing a framework in which such investigations can be conducted.
In some respects, EVMA draws inspiration from the work in computational evolutionary systems. However, most contemporary evolutionary systems are derived from older computing models that are not addressing the new challenges and opportunities of highly parallel architectures. These existing evolutionary computation designs are based on sequential tools and languages that were used in the previous era, when producing highly monolithic computing systems was the common practice (for a comparison and discussion of sequential and non-sequential computation see Section 4.11). For example, traditional genetic programming models are limited because they can only produce a single monolithic computing program. Even when parallel techniques are employed for the program generation phase, the actual result of the program space search is a single, sequential program.

To address these and other challenges and opportunities of the modern computing architectures, we decided to design, prototype and test a new model of massively parallel asynchronous computational architecture. Our model has been built from the ground up with automatic programming in mind and it tries to benefit from modern views of software and hardware architectures.

When developing our computational framework, we focused on two main objectives:

- The EVM computational model must be well suited for the automatic generation of massively parallel, autonomously interacting computational units, without the need for a human programmer at all; in other words, it should provide an architecture that can be used in a fully automated way, by automated program generators, to produce parallel programs and computation deployed on multi-core and multi-chip parallel architectures.

- The EVM computational framework should reflect, and thus intuitively model, naturally occurring phenomena; in particular, we are interested in modelling biological processes of life and evolution through a massively parallel distributed system of autonomous collaborating entities.

This fully automated computational model is the main feature that differentiates it from existing evolutionary computation models. Our framework provides programming constructs and features that depart from traditional well established programming models. Some of the EVM features are admittedly awkward for human programmers to use. These features provide, however, unique and expressive capabilities for automated program generators. We discuss some of these aspects below. We will make comparisons between the EVM architecture and traditional models of evolutionary computation in Chapter 6.

Computer science is founded and based on the notion of so called algorithmic computation, which is derived from the seminal work by Turing (1936–7). The main aim of classical computational models is to provide a comprehensive architecture that can be programmed by human programmers. This objective is most notably addressed by different algorithmic computation models. Alternatively, these can be represented by a class of computing machines inspired by the model of an abstract computing machine: a Universal Turing machine. We will discuss the Turing model in the following section. Later, we will present contemporary alternatives to the classical Turing machine model of
computation. We discuss briefly two other computing models proposed by Turing: choice-machines (c-machines), and oracle-machines (o-machines). We introduce the concepts of supertasks and supercomputing, and we propose a different, generalised notion of computation that tries to capture modern metaphors and perspectives on what computation is. We also briefly discuss and explain general ideas behind hypercomputing. We analyse basic assumptions on the nature of our universe. The purpose of this review is to present the basic issues and discuss why there is a heated debate between proponents and opponents of classical computational theory based on Turing machines.

4.3 Algorithmic computation

The word *algorithm* is derived from the name of the twelfth century Arabic mathematician al-Khwarizmi. Originally used to refer simply to the rules for calculating with numerals (i.e. Arabic numerals, which originated in India), the word gradually came to refer to any mechanical arithmetic procedure. In computation theory, the word means any mechanical procedure that maps numerical input into numerical output.

There have been several attempts in the early 1930s to formalise the concept of following a mechanical procedure, calculation, or an algorithm. The goal was to capture the intuitive process of computation as performed by a person equipped with a paper tape and pencil. The individual operations of a person were to be fully mechanised, i.e. the process must be described by some simple rules that anybody could follow (brainlessly so to speak).

One of the most influential figures providing a major contribution in the contemporary theory of computation is Alan Turing (Copeland, 2004). The most well-known contribution of Turing is to formally model the process of mechanical calculation (or *algorithmic computation*) performed by a human being equipped with a pencil and (an unlimited supply of) paper. Although Turing was not the only one working on such a formalisation, the Turing model is considered the simplest and the easiest to understand, and the Turing model of computation is well discussed in the literature (Hopcroft and Ullman, 1979; Papadimitriou, 1994; Li and Vitányi, 1997). The original name given by Turing to this particular model of computation is the *automatic machine*, or *a-machine*. Nowadays, it is known simply as a *Turing machine*.

Church and Turing, independently of each other, established different models of computation that were shown to share the basic assumptions mentioned above, and are in fact equivalent (see Section 4.3.2). One can say that they have equal computational power. What can be expressed and computed using one model of computation can be expressed and computed using one of the other models. What can be expressed in one model may require much more space to be expressed in another, and what can be computed may require substantially more time to be computed in one or another model. In principle, however, the models can express and compute the same set of algorithms, although, of course, from a practical perspective it is important to know exactly what the trade-offs of different models are. Theoretical computer science is concerned with the overall properties of classes
of algorithms. There is a sub-field concerned with resource-limited computations, and in this context, computer scientists investigate properties of different resource-bounded computation. Resources in computational contexts are effectively subject to space and time. Memory units (space) are the units necessary to store passive elements of any computational program (or algorithm). Processor ticks or clock cycles (time) are the units of processing dedicated to the execution of a single instruction.

4.3.1 Turing machines

There were several equivalent computational models proposed at the time of Turing’s work and in the following years (Gandy, 1988).

A Turing Machine, $TM$ (see Figure 4.1), consists of:

- A long paper tape, called a tape, which is divided into slots, one next to the other. Each slot can contain a symbol from some finite alphabet (a blank slot is usually represented as ‘0’). The tape is assumed to be arbitrarily extensible to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Slots that have not been written to before are assumed to be filled with the blank symbol.

- A moving head. A head can read and write symbols on the tape and move left and right one (and only one) step at a time.

- A state register, that stores the current state of the Turing machine. There is a finite number of states for any given Turing machine. There is a special start state with which the state register is initialised.

- An algorithm, or a transition function, that contains instructions for the head movements and read-write operations.

Through read-write operations, a head can change the contents of the tape. The state register remembers one of a finite set of states of the TM. The head moves and writes symbols on the tape according to a simple algorithm, that is prepared in the form of a set of rules, e.g.: “If your state is $S_1$ and the symbol you see is a 0 then replace this with a 1, move one symbol to the left, and set state $S_2$ as your new state.” This particular instruction could be written as a tuple: $S_1 01 Left S_2$ in the form usually used for the specification of for a Turing machine algorithm.

Turing proposed a special computing machine, called a universal machine, later named the Universal Turing Machine (UTM), that is capable of simulating any other Turing machine. In other words, the UTM can emulate any other machine of equal or lesser computational power\(^3\). The UTM can be used to show equivalence between different computational models: the trick is to write a program that emulates a given computing model on the UTM. The assumption of the universality of the traditional model of computation is known as the Church-Turing thesis (Gandy, 1988).

\(^3\)A discussion on hierarchies of progressively more powerful machines is presented in Section 4.7.5.
Finite number of states

Move right

Move left

TAPE

Tape reading/writing

Halt state

Finite number of states

Figure 4.1: Schematic view of a Turing machine.
4.3.2 The Church-Turing thesis

Before Turing formulated his notion of effective computability and argued for the unsolvability of the Entscheidungsproblem (Decision Problem) (Turing, 1936–7), Church (1936a,b) had proved a similar result. Church used the notions of recursive functions and lambda-definable functions to formally describe effective computability. Lambda-definable functions were introduced by Church (1932, 1941) and Kleene (1935). Recursive functions were introduced by Gödel (1934) and Herbrand (1932).

All these formalisms describe the same set of functions, as was shown in the case of functions of positive integers by Church (1936a), Kleene (1936) and Turing (1936–7). This result is simply known as a (weak) Church-Turing thesis.

The thesis is sometimes interpreted in a broader context, including and covering any intuitive notion of computability. The broader formulation of the thesis is sometimes referred to as a strong Church-Turing thesis. The strong thesis postulates that no other physically realisable model of computation at all can exceed the capabilities of Turing Universal Machine. The strong Church-Turing thesis has not been shown to be true or false, and it may never be resolved. However, following “Occam’s razor” principle, computational theorists assume it to be true (see for example the Wikipedia articles on the Church-Turing thesis and Turing machine). A good exploration of the thesis is provided in the Stanford Encyclopedia of Philosophy (http://plato.stanford.edu/entries/church-turing/).

Vergis, Steiglitz, and Dickinson (1986) proposed an analogue form of the Strong Church-Turing thesis:

Any finite analog computer can be simulated efficiently by a digital computer, in the sense that the time required by the digital computer to simulate the analog computer is bounded by a polynomial function of the resources used by the analog computer.

The resources of the analogue computer could be: time, volume, mass, energy, torque, angular momentum or other physical properties (Vergis et al., 1986).

The Church-Turing thesis stating the equivalence of all different models of computability based on positive integers is a fundamental concept in computer science and algorithmic information theory. The strong thesis is subject to certain assumptions concerning the nature of physical reality. We will discuss these assumptions briefly in the following section.

4.4 The Computable Universe

There is a difference between what is compact and expressive as an intuitive notion of computation, what can be physically realised at the moment given our contemporary technological capabilities, and

---

"One should not increase, beyond what is necessary, the number of entities required to explain anything". Occam’s razor is a principle attributed to the medieval philosopher William of Occam (Ockham). Also called the principle of parsimony.
what can be physically realised in principle. A detailed discussion on this subject is beyond the scope of this work. We only briefly clarify some of the misconceptions here. The true nature of the universe is not known and may never be known.

For instance, it is not known if the nature of our universe is inherently discrete or continuous. Hence, despite claims to the contrary (Zuse, 1970a; Fredkin, 1992), we do not know if our universe is reducible to the Universal Turing machine or not. Therefore, the validity of the strong version of the Church-Turing thesis cannot be established. Proponents of the strong Church-Turing thesis claim that our reality is inherently discrete, and is effectively reducible to a digital computer. In such a universe, non-computable processes would not exist, and there would be no source of true randomness in an algorithmic information theory sense (see Chaitin, 1987; Li and Vitányi, 1997). Any form of uncomputable input or oracles (in the Turing machine sense) would not be physically possible.

These views were clearly not intended by Turing himself. Turing never claimed the strong Church-Turing thesis. If asked, he would most likely have rejected the strong version of the thesis. In his seminal paper Turing (1936–7) introduced a choice machine (c-machine), that effectively models what would nowadays be called “interactive computing” (Wegner and Goldin, 1999). Clearly, it was Turing’s own belief that human operators could perform operations (computations) that are not captured by his own model of a-machines (or models of effective computation). Note that in Section 1 of his famous paper (Turing, 1936–7), Turing explains the concepts of a tape, machine, machine head, scanning, m-configuration and computable numbers.

If at each stage the motion of a machine (in the sense of 1) is completely determined by the configuration, we shall call the machine an automatic machine (or a-machine). For some purposes we might use machines (choice machines or c-machines) whose motion is only partially determined by the configuration (hence the use of the word “possible” in 1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.

Turing (1936–7)

There are two important ideas presented here by Turing. The first one is the notion of interactive computing that goes beyond that of an a-machine model. The second is the belief that even though an axiomatic system\(^5\) cannot provide proof for all possible propositions (due to an earlier proof by Gödel (1931)), a human operator could do so in these, so called, ambiguous configurations. One might speculate that Turing believed (as did many other contemporary authors) that humans could perform more powerful computations than those specified by mechanical algorithms (Bringsjord and Zenzen, 2003).

\(^5\)An axiomatic system is a logical system which possesses an explicitly stated set of axioms from which theorems can be derived.
It is not known if humans can be treated as a source of uncomputable input. However, it has been argued as plausible by some authors (Wegner and Goldin, 1999; Bringsjord and Zenzen, 2003) and the issue remains open. In the case that humans could be treated as a source of uncomputable input, a computer together with a human operator would represent a computational entity with computing power exceeding an ordinary Turing machine (the operator can provide a “choice” in the case in which a machine requests an input during its computation).

In the context of the EVMA, the nature of the physical universe is not of essential concern. The EVMA is designed to express many possible models of computing machines through massively parallel autonomous and interacting units. The EVMA tries to model in an intuitive and simple way many complex phenomena, and formal traceability is not an essential concern. In these respects, the EVM model departs from the mainstream of algorithmic computations. The EVM model relies on certain properties of the physical universe, and therefore it can, at least in principle, exploit certain properties that would lead to extended (beyond Universal Turing machine) computational capabilities. However, this is yet to be established empirically.

In the context of massively parallel asynchronous computing, where interactions happen autonomously between the computing cells and the environment, any formal analysis and formalisation of the computation by projecting it to an idealised working of a Turing machine must make a set of assumptions that will inherently violate the nature of the phenomenon being captured by the computing model. Because of this fact, and the general difficulty (or inability) of formal methods to capture complex systems dynamics and interactive behaviour, we think the Church-Turing thesis as being of limited applicability in the context of the contemporary theory of parallel and distributed computation, and the EVM architecture in particular. In principle, it may be possible (though tedious, prohibitively error-prone, and impractical) to provide a single view of a complex asynchronous and massively parallel system in the form of a Turing machine, given certain assumptions about the nature of the universe (see Section 4.4). However, for practical reasons, to obtain results and insights, simulation-based studies are more appropriate and may improve our understanding of given phenomena more quickly. Further research in the area of computability, especially in the field of interactive, massively parallel and asynchronous processing will provide the required insights and intuitions and may later lead to comprehensive formalisations of computational models and architectures.

The questions of computability or non-computability of our universe have implications for our scientific methods and philosophy in general. From the philosophical perspective, the nature of language and scientific pursuit is inherently Turing machine driven. The way computer scientists formally express and manipulate propositions can be modelled by a Turing model, and this may ultimately constrain and limit what can be formally comprehended and formally expressed. This suggests that any epistemological model must be inherently computable. This, however, does not mean that some other form of non-computable understanding may not be achieved. It is assumed and well acknowledged that human beings can reason in the formal axiomatic way. However, we do not pos-
sessed any formal methods to test if human beings can possibly also reason in non-axiomatic, intuitive and uncomputable ways. This argument is most notably pursued by Bringsjord and Zenzen (2003), which specifically argues that human minds can work as super-computers (the notion of super- and hyper-computing will be discussed in the following section).

### 4.5 Supertasks

\[ \ldots \text{if everything that exists has a place, place too will have a place, and so on ad infinitum.} \]

[Aristotle, Physics IV:1, 209a25]

In this section, we will provide other necessary contexts, to discuss some of the main elements and arguments related to hypercomputation. When discussing hypercomputation and extended models of computation (i.e. beyond the Turing machine computational models) one element is called upon frequently: *infinity*. The reader should remember that the term *infinity* has a distinct and precise meaning in mathematics that is somewhat different from the colloquial usage (and therefore common intuition). The word *infinity* comes from the Latin word *infinito* and simply means unending. In mathematics \( x \to \infty \) denotes \( x \) growing beyond any assigned finite value.

The basic intuition for Turing machine based computation is the notion of an automated and mechanical process of calculating an expansion of a real number. All real numbers that can be represented as a particular instantiation of a Turing machine are therefore called *computable numbers*.\(^6\). The contemporary intuitions regarding computable numbers, however, restrict the computation to be finite. For example, the expansion of \( \pi \) can be computed (i.e. there is a short algorithm for calculating the successive digits of the expansion of \( \pi \)) and thus we might intuitively consider it to be computable. However, because we cannot execute an infinite number of steps in a finite time, even though \( \pi \) should be formally considered as *computable*, usually, it is considered *non-computable* due to practical limitations. To compute \( \pi \) would require an infinite number of steps. Again, depending on the nature of the universe itself, such a computation may or may not be physically possible.

There are some properties of \( \infty \) that are somewhat different to normal finite arithmetic, and to fully understand the implications of the concept of infinity is not easy. A good discussion of the concept of infinity is provided by Moore (1990). Moore also presents all the well known paradoxes relating to infinity, for example, Hilbert’s Hotel\(^7\) and The Thomson Lamp\(^8\).

In the context of virtual machines, we will only distinguish between countable (enumerable) infinity, and uncountable (non-enumerable) infinity. Unless specified otherwise, we will mean countable

---

\(^6\)Hence the title of Turing’s seminal paper (Turing, 1936–7).

\(^7\)There is a hotel with an infinite number of rooms. Even when the hotel is full, it can take one, or even an infinite number of new guests in (e.g. by shifting everyone already in the hotel to another computable room).

\(^8\)The lamp is either on, or off. For the first half minute, it is on, then for the next quarter minute, it is off, then for the eighth of a minute it is on, etc. for an infinite number of times. After the minute lapses, is the lamp on, or off?
infinity. Infinity in the context of virtual machine operations introduces an important concept: a supertask. A supertask (Thomson, 1954-55) is a task that takes a countably infinite number of steps that are all being executed (or carried out) in a finite amount of time. A hypertask is a task that takes an uncountably infinite number of steps, also executing them all in a finite amount of time. Accordingly, a supermachine is a machine capable of performing supertasks, and a hypermachine is a machine capable of performing hypertasks. In this work, we do not pay attention to distinguishing between these two types of infinities or these two types of hypermachines (a supermachine can be viewed as a special case of a hypermachine, and a Turing machine can be viewed as special case of a supermachine). For clarity, we will use the term hypercomputation equally for supermachines and hypermachines (which is a common practice in the literature).

The concept of a supertask is used to describe (and understand) some of the paradoxes based on infinities such as those exemplified by Zeno’s Paradox, the Thomson Lamp, and Hilbert’s Hotel (Moore, 1990). At the same time they are a subject of some controversy in mathematics and philosophy. There is no agreed resolution as to the logical consistency of the concept of infinity, and some authors argue that due to contradictions associated with infinity, the concept of supertasks is logically impossible. Others argue otherwise and hope that the study of supertasks can help us improve our understanding of and theories about the physical world. In particular, Moore (1990) claims there is an inconsistency in the concept of infinity and supertasks. In contrast, Bringsjord and Zenzen (2003) argue that Moore is wrong, that the notion of supertasks is mathematically sound, and it is consistent with standard logic and basic mathematics.

What is not resolved (and may not be resolved in the near future, or ever) is the question of the physical realisation of a machine capable of performing supertasks. There are some fundamental problems in setting up experimental tests for supertasks or hypercomputation. Some argue that such experiments have already been successfully conducted, and they point to advances in proof generation made by mathematicians in their heads. It is not clear however, if a given proof invented by a mathematician is in fact an uncomputable one.

It has been argued (Bringsjord and Zenzen, 2003), that humans are capable of hypercomputation, through their capabilities of manipulating abstract concepts, in particular infinite sequences of numbers. Although these arguments are speculative as to the physical reality and the possibility of a realisable hyper-computer, the concept of hypercomputation seems to be accepted by many in the mainstream research community. Some argue that certain physical devices can exhibit hypercomputation, or that human beings can perform it in their minds (Bringsjord and Zenzen, 2003; Aaronson, 2005). There has been, however, so far no conclusive and verified experimentation on hypercomputing. Any such positive result would have an enormous consequence in computer science, physics and philosophy. The detailed discussions of the implications are beyond the scope if this work, and must be left for further investigations (further discussion is provided in Appendix C).
4.6 Hypercomputation

Hypercomputation is a general term that covers a broad range of computational models. Usually, it means any computation that goes beyond that defined by the Turing machine model. However, a Turing machine itself can be considered to be a special case of hypercomputation. Models that go beyond the Turing model are also known as super-Turing, non-standard or non-recursive computation (Teuscher, 2006; Aaronson, 2005; Bringsjord and Zenzen, 2003; Copeland, 1997). Although hypercomputing goes back as far as Turing, it has recently become a subject of increasing interest and has emerged as a relatively new, multi-disciplinary research area, spanning a wide variety of fields: computer science, mathematics, philosophy, physics, biology and others.

Physical analogue computation is often brought up when discussing physically plausible hypercomputing models. Analogue computing usually means computation performed by any deterministic physical device that uses a fixed number of physical continuous (thus real-valued) variables to represent each problem variable (Vergis et al., 1986). In other words, internal state is represented by continuous variables instead of discrete ones (e.g., soap bubbles, protein folding, gears, black holes, etc.)

Although originally not designed with hypercomputation as an objective, our EVM architecture provides certain properties that distinguish it from traditional discrete algorithmic virtual machines, and bring it to the realm of super-Turing architectures. There are three fundamental properties of the EVM architecture that satisfy this:

- there is interaction, communication and collaboration between computational cells; EVM consists of a massively concurrent architecture: all cells compute autonomously and asynchronously;
- high scalability – there is no inherent limit to the number of interconnected computational cells;
- evolvability – all cells can autonomously change the computation they perform; all interconnections between cells can dynamically change; cells can disappear and appear dynamically.

In this connection, the near-term feasibility of realisation of a supermachine is not important. What is important, however, is the distinction between machines that are not theoretically capable of hypercomputation (computation above the Turing limit), if such computation is physically possible, and machines that, if it is shown that hypercomputing is possible, would be capable of performing super- or hypertasks.

For example, a simple working model of a single-tape, single-head Turing machine is capable only of Turing-level computation. Regardless of any properties of physical reality, or physical realisation of such a computing model, such a machine would not be capable of hypercomputing even if hypercomputing were to be physically possible. However, the computational power of multi-tape, multi-head Turing machine depends on the synchronisation properties and the communication between the heads and tapes. In situations where there is discrete time synchronisation between the
read/write operations of the heads, the multi-tape model is equivalent to a single-tape Turing machine (Papadimitriou, 1994, Theorem 2.1). However, in the situation where there is no synchronisation, and there are possibly continuous (or discrete, but with infinitely small time ticks) time delays between the operations of different heads, the multi-tape model exceeds the computational power of a Universal Turing machine (Siegelmann, 1995; Wegner and Goldin, 1999).

The EVMA can be theoretically modelled as a hierarchically organised multi-tape multi-head Turing machine computational model, although due to the massively parallel and asynchronous nature of the computation, there are no practical parallels between these two models. These two models however share the following essential property: in an uncomputable universe, the EVMA would be capable of non-algorithmic computation. In other words, it would be capable of computations that would not be possible when carried out on a TM.

4.6.1 Is hypercomputing possible?

As already mentioned, Turing machine computation is believed by some to be the only possible computing model (see, for example, the following discussion on the Computational Complexity weblog: http:// weblog.fortnow.com/).

“We may summarize: the positive evidence provided by enthusiasts for hypercomputation amounts to no more than the trivial remark that given a physical “oracle” that somehow makes uncomputable information available, it will become possible to compute other uncomputable functions as well. The great success of modern computers as all-purpose algorithm-executing engines embodying Turing’s universal computer in physical form makes it extremely plausible that the abstract theory of computability gives the correct answer to the question “What is a computation?” and, by itself, makes the existence of any more general form of computation extremely doubtful.” M.Davis, “The Myth of Hypercomputation”, p.209.

This remark is precise and correct regarding the physical oracles and the sources of uncomputable data. However, oracles and source of uncomputable data are only part of the issue. The remark above is missing an important point: a Turing machine is limited to a certain class of computation, regardless of the nature of the universe, and the availability of uncomputable input. Also, modern computers are not entirely based on the Turing universal computer:

- computers do not have unlimited data storage;
- computers do not take all the input at the beginning of the computation and calculate the output in isolation;
- and, computers work in parallel with other computers and with human operators, in an asynchronous fashion.
Regardless of the arguments about the nature of the universe and how much is algorithmically possible (see Section 4.4 for discussion), it makes sense to use models closely mimicking the actual physical nature of computers, and to provide intuitive notions that can be used to understand, program and manage the complexity of real-world interactions between different computing devices (including human operators). Due to these reasons, alternative models of computation, especially those focused on parallel asynchronous computation, are valuable to progress our understanding in information processing and virtual machine research.

4.7 Massively parallel asynchronous computation

4.7.1 Preliminaries and motivation

One of the characteristics of the field of artificial life and artificial computational evolutionary systems is the practice of modelling and experimenting with systems composed of a large number of small, concurrently executing and asynchronously communicating computational systems, or agents. We refer to such a system as a massively parallel asynchronous system. Many natural phenomena can be characterised by the interactions of a large number of asynchronously and concurrently executing systems. Properties distinguishing them from traditional parallel computational systems and architectures are (A) the fact that the subsystems communicate asynchronously, without any global synchronisation mechanism, and (B) the large scale on which such interlinked networks of agents operate.

To investigate the proposed model, we first look into possible formalisms that can be employed when discussing and analysing such models. First, we concentrate on asynchronous communication aspects, and discuss interactive machines. Later, we provide our own notion of computation, which is well suited for the purpose of modelling highly distributed and asynchronous computation.

4.7.2 Interactive machines

Interactive machines are an intuitive notion of computation based partially on Turing machine models, and interactive computation models have been pursued and formalised by many contemporary computer scientists. One of the earlier notable contributions is the model provided by Milner, called Calculus of Communicating Systems (CCS) (Milner, 1980) – a process calculus where actions model indivisible or atomic communications between exactly two participants. The formal language includes primitives for describing parallel composition, choice between actions, and scope restriction. CCS has been a basis for the extended and dynamic interaction formalism called the $\pi$-calculus (Milner, Parrow, and Walker, 1992; Milner, 1993). The $\pi$-calculus is a mathematical model of concurrent processes. The central notion is that of interaction via message passing, and the model assumes that interconnections can dynamically change during the computation. This model is used in the area of multi-agent systems (MAS). Another known model that proved to be successful in mod-
elling complex distributed, concurrent, and networked systems is the Input/Output Automata (IOA) introduced by Nancy Lynch and Mark Tuttle (Lynch and Tuttle, 1989). Peter Wegner and Dina Goldin popularised the general notions of interactive computing through their work on Persistent Turing Machines and Multiple Interactive Machines (Wegner and Goldin, 1999, 2003) – these are discussed in the two subsections below.

Interactive computing is believed by some to go beyond the Turing limit (Wegner and Goldin, 2003), although this result depends on certain assumptions about the nature of the universe. In particular, the notion of computable or non-computable information sources and time delays (see Section 4.4). Regardless of the nature of the universe, however, interactive computing tries to capture the intuitive notion of expressiveness based on the capacity of an observer to make discrimination among different computations.

Note that all the discussion about the nature of computability is provided here in the context of elucidating our EVMA. We are not investigating general properties of resource-bound or generic computing systems. Instead, we are confining our focus to more narrow architectural implications of certain assumptions about the nature of discrete computation, and so we are not presently pursuing the potential of hypercomputing capabilities of the EVMA as such, although, it may become a subject of further investigations in the future.

4.7.3 Persistent Turing machines

To capture and formally express the notion of interactive behaviour, Persistent Turing Machines (PTM) have been proposed (Wegner and Goldin, 1999). A PTM is a minimal extension of a Turing machine: it is a multi-tape machine with a persistent work tape that is preserved between interactions with the computing machine (that is, various programs that are being executed on the machine use the same tape, and the state of the tape is not reset between these programs). Such a machine can model services over time provided by a persistent, object-oriented or other type of reactive systems. A PTM provides a natural model for single-user databases. It has been conjectured that PTMs can model systems that cannot be expressed as computable functions (Wegner and Goldin, 1999, 2003; Teuscher, 2006), although these claims are subject to heated discussions and controversies. It has been argued, that PTMs are well suited to model sequential interactive behaviour (Wegner and Goldin, 1999), where the computing device or agent evolves as it processes the inputs. This evolution is represented by the change in the contents of the PTM’s work-tape – the PTM output is a function of both the input and the state of the evolving work-tape.

Regardless how expressible the proposed model is, it provides an interesting perspective on possible computational architectures that are built around the concept of interaction and history-dependent computation. It provides an alternative formalisation that may be more appropriate for the kind of computation performed by a modern computer. Also, in the context of life-long reinforcement learn-
ing, this is an attractive model.

4.7.4 Multiple Interactive Machines

The natural extension of the PTM model discussed above is the Multiple Interactive Machines model, MIMs (Wegner and Goldin, 2003), see also (Teuscher, 2006, pages 159–194). The idea behind MIMs is based on the observation that multiple asynchronously-interacting agents provide a much richer computational environment than the one provided by the sequential Turing machine. It has been argued that asynchronous interactions between autonomous entities are capable of super-Turing computation (Bringsjord and Zenzen, 2003; Teuscher, 2006). The idea is to remove the strict discrete computable aspects of algorithmic computation and to provide a model that is a (potential) mix of oracle machines (o-machines) and analogue computation (asynchronous interactions in a continuous time universe). The trick is not actually to make any assumptions about the availability of analogue computability or/and access to oracles. If real value computing is not available (i.e. the nature of time is not continuous) and if oracles are not accessible (all input streams are computable), interactive computing is equivalent to a TM. But if the nature of the streams “allows” oracles or continuous time delays, then interactive computing “works” better than a TM. Thus, opening up the computing system to interact with the real world allows certain properties that are not possible through the pure TM model.

Developing massively concurrent asynchronous systems is an important and difficult area, because these systems are capable of emergent and complex behaviour that go beyond the complexity of traditional systems. The field of multi-agent systems is one of the areas of computer science that tackles that issue. MASs utilise decomposition and well-known established intuitive notions from the realms of the physical and social sciences. The notions of agenthood and its associated ideas of autonomy, asynchronous unreliable message-passing communication, interaction, and societies and institutions are being adopted and used throughout this research arena. A related area that addresses similar issues is the computing research area concerned with hypercomputation. In particular, the multiple interactive machines (MIMs) paradigm (Wegner and Goldin, 2003) and work on the Evolvable Virtual Machine architecture (Nowostawski et al., 2004) parallels MAS research. To some extent these fields share similar notions and modelling abstractions, such as asynchronous communication and interaction.

4.7.5 Algorithmic hierarchy

One of the established modelling power hierarchies extending up to the Turing limit is the Chomsky Hierarchy (Chomsky, 1956, 1959).

---

10 Whether the nature of time is continuous, and whether there are oracles (some argue that input provided from humans is uncomputable, for example) is another story, which can be pursued on a personal preference basis and is not critical to the discussion about the expressiveness of the computing models.
The power of a given machine is expressed through the use of a concept of a formal language (and formal grammar) that the machine is capable of recognising (or accepting). A formal grammar consists of a finite set of production rules that describe a particular language. The rules are written in such a way that the left-hand side, which must match a condition in the environment for a rule to be applied, is substituted by the right-hand side. The right-hand and left-hand sides consist of some terminal and non-terminal symbols. The non-terminal symbols denote production rules. A rule may be applied to a word by replacing an occurrence of the left-hand side by the right-hand side. A derivation is a sequence of rule applications. Such a grammar defines the formal language of all words consisting solely of terminal symbols that can be reached by a derivation from the start symbol.

- At the bottom of the hierarchy (with the least computing power) is a computing machine based on (or equivalent to) a finite state automaton (FSA) capable of accepting (or recognizing) regular languages. A regular language is a language that is expressed by a regular grammar, and there exists a finite state automaton that can accept such a language. A regular grammar is either left regular or right regular. A right regular grammar is characterised by production rules of the form $A \rightarrow aB$, where $A$ and $B$ are non-terminals, and $a$ is a terminal. Accordingly, a left regular grammar has rules of the form: $A \rightarrow Ba$.

- The next level in the hierarchy comprises push down automata – these have the same elements as a FSA, but also include a stack. They accept context-free languages, expressed by context free grammars. Context-free grammars have production rules of the form $A \rightarrow \gamma$, where $\gamma$ is a string of terminals and non-terminals, and $A$ is a non-terminal.

- The level above push down automata consists of linear bounded automata. A linear bounded automaton is just like a Turing machine, but the tape accessible for computation is restricted to a finite length (a constant multiplied by the length of the input). A linear bounded automaton is capable of accepting a context-sensitive language, whose grammar is based on production rules of the form: $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $\alpha, \beta, \gamma$ are strings of terminals and non-terminals, and $A$ is a non-terminal symbol.

- The most powerful level, unbounded Turing machines, can accept recursively enumerable languages, whose grammars can have productions without any restrictions. In particular, note that unlike other types, grammars for languages accepted by Turing machines may have production rules $S \rightarrow \epsilon$, with $\epsilon$ is an empty string, where $S$ can appear on the right hand-side of another production rule.

Everything expressible by a regular language is expressible by a context-free one; everything expressible by a context-free language is expressible by a context-sensitive language, and everything expressible by a context-sensitive language is expressible by a recursively enumerable language. These are all proper inclusions. Table 4.2 summarizes this.
The hierarchy provided above is well-suited for the traditional theory of computation; however, it does not naturally generalise to more powerful machines – machines exceeding the Turing limit. Such extended classifications have been proposed elsewhere (Bringsjord and Zenzen, 2003), and we will concentrate our discussions on the level above the Turing level – the first hierarchical level of hypercomputing. In the following sections we will introduce the basic concepts of hypercomputation, we will discuss some proposed models of hypercomputing machines, and we will speculate on the possibility of physical realisations.

### 4.7.6 Interactive hierarchy

In analogy to finite state automata and the language/grammar hierarchy of algorithmic computation presented above, a similar expressiveness hierarchy can be constructed for interactive computation (Goldin, 2000; Wegner and Goldin, 2003).

A different approach based on the *behavioural equivalence classes* is also proposed by Goldin (2000). This work uses the notion of stream-based interactive machines. We distinguish between the interactive machine and the environment in which this machine is embedde. The stream of data flows from the environment into the machine, and the answers flow back from the machine into the environment. An environment $O$ will induce a partitioning of all machines into equivalence classes, where the members of each class appear equivalent in $O$. An environment $O_1$ is said to be *richer*, or *more expressive*, than $O_2$ if it contains more equivalence classes. It has been demonstrated (Wegner and Goldin, 1999) that TMs are a proper subset of PTMs, and PTMs are a subset of MIMs.

Interactive computing in a Turing computable universe is equivalent to Turing computation. It is isomorphic to the notion that a multi-tape multi-head Turing machine is equivalent to the single-tape Turing machine (following (Papadimitriou, 1994, Theorem 2.1)). However, in a non-computable universe, the multi-tape model would potentially exhibit non-computable properties, and the equivalence property does not hold anymore. This is because the asynchronously communicating processes do not necessarily have to follow computable time differences. Therefore, the synchronisation intervals between asynchronous processes may become a source of uncomputable input, i.e. an oracle.
4.8 Generalised computation

It is not currently known precisely what the general notion of computation should be, because it is not known what the nature of our universe is. It has been argued, however, that the strong Turing thesis is not applicable in general to contemporary computation, and that the simple Turing-machine based models of computation are not the best models for the essence of computation (Wegner and Goldin, 1999; Bringsjord and Zenzen, 2003; Goldin and Wegner, 2005; Akl, 2005).

There is ongoing work in finding better models that are capable of expressing all the possible computations that current computing devices can perform. It has been argued that other alternative models of computation are needed to extend general notions originally developed in the 1930s. In particular, as discussed above (Wegner and Goldin, 1999; Goldin and Wegner, 2005), it is argued that interactive computing models are more appropriate to modelling computation performed by contemporary computing devices. The interactive computing models break certain assumptions of algorithmic computation, most notably, the communication with the outside world during the process of computation. This goes back to the original notion of e-machines proposed by Turing. On the one hand, considering human operators together with many other independent and autonomous entities seems like an essential extension to our modern models of computation. On the other hand, this will lead to intractability and we will lose some of the formal properties that the abstract Turing-machine model offers.

4.8.1 Algorithmic abstractions

Despite its huge popularity, the algorithmic model of computation relies on certain properties that somewhat constrain its applicability in the context of modern computing devices. One such property, which makes it easy and tractable for formal analysis, is that there is no interaction with the outside environment during the course of computation. In other words, the computation starts by writing all the input that is necessary to perform the computation onto the tape. After that, the computation is carried on without any input from the outside world. The second feature of the TM model is that it is assumed that there is no cost in reading the initial marking of the tape. The assumption is that no matter how big the initial marking is, it can all be read in a single operation that takes no time. There is no cost of or physical constraints on these two properties. Once the data is read instantaneously, the machine carries out its algorithm, without any contact with the outside world.

The original view of a metaphorical machine carrying on computation has been replaced by physical computers, immersed in the real world, continuously interacting with human operators and other computing devices. Computing machines are not just abstract devices, they are physical machines with properties, that unquestionably vary from the abstract model of a TM. For example, the tape, or memory storage, is never infinite in capacity. The computation can take input and produce output during the course of its “life”. A given computation often requires input from the outside world, during the course of computation. Modelling such computations with a TM would be, at least, problematic
and cumbersome, and at worst, impossible.

Needless to say, there are many inherently nice properties of the Turing model: it is simple, abstract, and it is amenable to formal analysis, predictions and studies. Studying abstract properties of Turing computation yields many insights into computer science and complexity theory.

But the TM model, although conceptually simple and attractive, is somewhat displaced from the contemporary notion of computing devices that we are so accustomed to. TMs were meant to be a thought experiment to investigate the limits of mechanical computation. Perhaps a more natural form of computation is a simple notion of transforming an input into an output. A TM does not accept external input when it computes and this fact prevents it from being applicable in modelling natural processes. Contemporary computing machines do not naturally fit into the TM model. Computations inherently involve interactions and communications with the program by an external agent (a person or another program). Interaction machines are natural extensions and generalisations of TMs.

### 4.8.2 Computing machines

We will use the term *computing machine*, or simply a *machine*, to be a metaphor for an entity that has the following features\(^\text{11}\):

1. it can read inputs from the outside environment;
2. it can perform a sequence of *calculations* on the input data, and provide the results delivered as the output;
3. it has an internal memory and can store data read from the input internally, or intermediate results of calculations;
4. data provided by the output is the result of the input data, the internal state of the machine, and the calculations; i.e. the output is always obtained in a mechanical, deterministic way through the calculations.

In other words, the machine has the ability to communicate with the outside world by means of some input/output channels, has the ability to internally store the state of the computation, and has the ability to perform basic arithmetic operations on the input data. Intuitively, one can think of a *digital computer* as a physical realisation of a particular *machine* that computes digital data. *Computes* means any combination of the following: reads in, stores, transforms, and writes out. A machine (or a digital computer) takes digital input from the outside world, stores and manipulates this input and generates digital output (e.g., in the form of a digital image on a monitor, or in the form of a sound).

Note that a TM is a special case of a machine, where the interaction with the external environment is limited to the initial marking of the TM’s tape and reading the final state of the tape.

\(^\text{11}\)This formulation is inspired by the discussion presented by (Akl, 2005).
Depending on the nature of the data (e.g. finite, digital, continuous, analogue), the nature of the computation (e.g. having a finite number of steps in each operation, or an infinite number of steps) and the nature of interaction with the environment (e.g. interaction only at the beginning and at the end of computation, or a continuous stream of input/output data during the computation) one can build a different taxonomy of machines. In this work we focus on finite, interactive machines.

4.8.3 Realisation of computing machines

Most of the current computational machines realised as physical devices follow the above model. However, there are some characteristics that are typically not taken into account when discussing abstract computational machines. For example, in an abstract TM model, all the input to the process can instantaneously appear on the machine tape. There is no delay, nor problem, regardless of the actual data size that must be placed on the tape. This presents some problems when it comes to actually building a physical realisation of the abstract computing model.

Akl (2005) argues that all physical devices have a limit of how much data they can read from the environment in a single step; therefore, no physically realised computing device can simulate another device that is capable of more operations per step.

Consider an example of a computation accessing \( n \) real world variables that vary over time. Given a computing machine that can read only \( n - 1 \) variables in a single operation, such a machine would be unable to perform our desired computation. This is because the computation would require two time steps to read all \( n \) variables – in which case the values of some of the variables could have changed between the first and second time-steps. In such a scenario, an \( n \)-tape TM is more powerful than a single-tape TM, because a single-tape TM can only read a single variable in a time step, whereas an \( n \)-tape \( n \)-head TM can perform all \( n \) operations in a single time unit. A detailed discussion regarding the equivalence of different finite machines is presented by Akl (2005).

4.8.4 Algorithmic complexity

Computational complexity theory (Papadimitriou, 1994) is the branch of the theory of computation that studies the resources, or cost, of the computation required to solve a given computational problem. This cost is measured in terms of abstract parameters, such as time and storage space. Time represents the number of steps (primitive instructions) it takes to solve a problem and space represents the quantity of information storage. There are certain trade-offs between time and space, and in general one can increase the speed of processing by increasing the memory usage (Jones, Gomard, Sestoft, and Andersen, 1993). Space requirements can be profiled over time, especially in consideration of a multi-user computer system. In the context of parallel processing, the notion of parallelisable time is often considered. It provides a limit to how far the computation can be parallelised.

Algorithmic complexity (Li and Vitányi, 1997) is a theory of complexity based on the notion of the length of computing programs generating output sequences. In short, the complexity of any given
sequence is directly proportional to the length of the shortest program generating that sequence. It follows directly from the definition of algorithmic complexity that no computer program can generate output more complex than the program itself. This provides an upper limit on the complexity of any computing program that does not communicate with the outside world during the computation.

To provide continuous complexity growth, a continuous stream of complexity must be provided to the computational device. In other words, communication with the outside world is essential in open-ended evolutionary systems. This is an important result, which is sometimes overlooked in the field of evolutionary computation. Any closed computation can be expressed, for example, as sorting, and as such is unable to generate a growing level of complexity characteristic of biological life.

4.8.5 The No Free Lunch Theorem

"There Ain’t No Such Thing As A Free Lunch" (TANSTAAFL) is a phrase popularised by science fiction writer Robert A. Heinlein in his 1966 novel, *The Moon Is a Harsh Mistress*, which discusses the problems caused by not considering the eventual outcome of an unbalanced economy. This phrase simply means that a person or a society cannot get something for nothing. Even if something appears to be free, there is always a cost to the person or to society as a whole even though that cost may be hidden or distributed.

![Figure 4.3: An illustration of the no-free-lunch theorem. The red line shows the performance of a highly specialised algorithm and the blue one a general-purpose algorithm. Note, that both algorithms perform on average equally well as they are applied to different problems.](image)

The no-free-lunch theorem (NFLT) is a theorem in the field of combinatorial optimisation developed by physicists David H. Wolpert and William G. Macready (Wolpert and Macready, 1995, 1997), inspired by the phrase TANSTAAFL. The NFLT states that:

> [...] all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions.

(Wolpert and Macready, 1995)

Over the set of all possible problems, each search algorithm will do on average as well as any other (demonstrated graphically in Figure 4.3). This is due to the bias in each search algorithm, because sometimes the assumptions that the algorithm makes are not the best ones for a particular input. “A
general-purpose universal optimization strategy is theoretically impossible, and the only way one strategy can outperform another is if it is specialized to the specific problem under consideration” (Ho and Pepyne, 2002).

In other words, the theorem states that no single algorithm can outperform random search, if all possible problems are taken into consideration. It is relevant when comparing different algorithms on different test cases and test problems. In the context of evolutionary computation, genetic algorithms and simulated annealing, the theorem is often used against claims that a given method that does not use domain-specific knowledge outperforms other no-domain-specific-knowledge algorithms. For the general case, with all possible test cases, such claims are unjustified. A given algorithm outperforms another, only in a finite clearly defined subset of all possible test cases, and only if it successfully employs custom domain-specific knowledge. For practising engineers and other optimisation professionals, the theorem justifies the view that as much prior domain knowledge should be utilized as possible, and custom optimisation routines constructed for particular domains and problem instances should be employed.

In the context of life-long reinforcement learning, and optimal search procedures, the theorem makes clear that the search bias of any fixed search procedure for a given problem will be outperformed for some range of inputs by the search bias of another algorithm. By bias, we mean here an influence in a particular direction, or a prejudice of the search mechanism. To achieve long-term optimal search, a bias must be a subject to continuous tuning and modification. That means the problem encoding, the computational language used to compute the search, the solution space, parameter space, etc., must all be subject to reinforcement learning and modification. The optimal search method must be able to reflect on, reify and modify the initial search bias.

This is what some of the optimal search mechanisms try to achieve (Levin, 1973; Hutter, 2002; Schmidhuber, 2002a, 2004) through bias-optimal search. This is also what we investigate through our EVM architecture: different aspects of bias-optimal search and trial-and-error machines. The EVM provides special architectural constructs to manipulate the bias as outlined in Chapter 3.

In the following section, we discuss the formal definitions related to multi-level processing. The architectural details will be described later – here we only focus on the formal analysis of the proposed hierarchical computation model.

### 4.9 Hierarchical Computation

In system science and cybernetics any system under investigation is thought of as a composition of multiple subsystems, each of which can itself be decomposed into subsystems, and this follows all the way down to a basic, fundamental level (Simon, 1968). Hierarchies help us to deal with complex phenomena by decomposing them into more manageable subsystems and investigating the interactions between these subsystems, one interaction at a time. The emphasis is placed on investigation of properties on different levels, mutual dependencies, and interactions between and within the hierar-
chyla levels. Hierarchical decomposition of the problem space deals with complexity in a way that is natural and intuitive to humans.

Many naturally occurring phenomena can be modelled (to an arbitrary precision) by a hierarchy. Building such a hierarchical model is not an easy task, especially if there are a number of interrelated levels involved, and if we do not know all the constraints on the subsystems in advance. Nevertheless, the advantages of hierarchical models, especially for human comprehension, are substantial. For example, human society is composed (in a simplistic view) of institutions, which are composed of individuals, which are composed of organs, which in turn are composed of tissues and cells, which are further decomposable into chemical reactions and physical processes. We would not be able to understand various relationships between reactions at molecular levels and reactions at the level of institutions (like company mergers or bankruptcy). Such relationships are implicit and difficult to deal with directly. They become manageable through hierarchical models. A computer program is another example of a hierarchical system. Instead of programming computers by means of rearranging the physical wiring between Boolean gates inside digital computers, programmers write software in higher level programming languages by using compilers or interpreters and virtual machines to logically perform the re-wiring at the (logical) Boolean gate level. These higher-level programs are compiled and executed on lower-level machines, which in turn are executed on specific Boolean gate implementations (hardware). Again, it would be unmanageable to deal with all the higher-level programming abstractions like objects, methods, and attributes in the object-oriented paradigm if it were mapped directly to physical Boolean gates. All the intermediate levels are necessary in practice.

4.9.1 Levels of organisation

Some believe that all sufficiently complicated systems are modelled best by hierarchical models (Holland, 1992; Rosca, 1997; Spector, 2002). The formalism discussed here aims at providing a uniform framework for modelling dynamic hierarchies, and is inspired by the artificial chemistry model of Mayer and Rasmussen (1998).

When talking about life it is often tempting to refer to emergence. When the parts of some system become coherently organised and might be characterised as being something new, we often say that a new level of hierarchical organisation emerged (Jones, 2002). Talking about hierarchical levels of organisation, describing boundaries between levels and understanding how a collection of constituents can actually become a new level (Jones, 2002) is not yet fully understood or completely formalised. There are however some existing proposals (Simon, 1968; Groß and Watson, 2002) which are becoming more and more useful when dealing with and understanding the phenomenon of life.

4.9.2 Interpretations and organising relations

One of the fundamental tasks in research on complex systems is to understand emergence and the means by which complex objects can come into existence from the interactions of their constituent
parts. To help with this task, a model relying on a particular system of *organising relations* has been proposed by Jones (2002). The idea is to look for the low-level laws and rules that the constituent parts must obey. The emerging levels then are a (natural) consequence of a given configuration and these low-level laws. For example, in some perspective the physical laws of nature are the organising relations for the reality we observe around us. The level of atoms, the level of organic molecules and cells, the level of organs, organisms, and the level of societies, etc., are all subject to the lowest level physical laws, and can be seen as derived from them. All the emerging phenomena are either inherently part of the lowest level laws, or are inherent in the interactions between lower level entities.

To better understand the hierarchical organisation of a system, consider the following example. Let us assume we have a deterministic discrete virtual computational machine (Nowostawski, 2002), implemented on some electronic hardware, e.g., a personal computer. The memory is binary, and the processor operates in binary. Imagine that we can observe all the memory cells’ states and the current processor state in an instant. Imagine now, that this virtual computer is “running” some sort of application, and our task is to understand what this application is, how it is using the memory and the processor, and what it is showing on the screen (this being simply a memory image as in standard video cards). Once we know that, then by modifying the memory cells’ states we would be able to control and use this application. Is that task feasible? It depends on the scale. If the processor has many different states and the memory space is huge, we would not be able to deal with the emerging complexity at all. However, if the processor only has several possible states, and the memory is small, we would be able to build a complete model of all the transitions and states of a running process.

Given such a model, would we really “understand” what the application was all about? Suppose the actual application is a word processor. Would it be possible to derive understanding from the complete binary map of all the transitions? Partly due to the fact that the higher-level context is unknown it would not, in fact, be possible. We could only observe some of the patterns and regularities of the process, and we could build some more or less accurate abstract models of what is going on within the process. However, we would be unable to know what the actual application is. We may build a model that is close, or not close at all, but gives the same results with an assumed level of accuracy. Similarly, looking at the low level laws of cellular automata, we cannot predict and derive all of the possible emerging structures and interactions, even though they are “encoded” in these low-level laws.

---

12 Some researchers maintain the claim that a truly reductionist programme could be, in principle, successfully carried all the way to the fundamental physical level. We do not claim that this is necessary (or useful), and in fact, we stress the dependencies between different levels too.
4.10 Virtual machines

4.10.1 Motivation

Hierarchically organised recursive virtual machines can be used as a computational modelling technique. Such a technique is based on the notion of computing machines introduced earlier (see Section 4.8.2). The discussed technique allows an analysis of different aspects of hierarchical complex system decomposition, together with the analysis of interactions between and within different hierarchical levels. This may help to understand a modelled problem or phenomenon better, giving us at the same time robust and adaptable computing framework. In this section we present a formal introduction to hierarchical virtual machines, we compare the proposed model with other models from the field of hierarchical evolutionary computation, and we provide some simple examples (using Boolean algebra) as a demonstration of basic principles. An approach to code generation called software growing is proposed and discussed later in this chapter.

The main objective of hierarchy-based models is to analyse and exploit the interactions between different levels. The additional objective for dynamic hierarchy-based models is to construct all necessary hierarchy levels dynamically, giving extra flexibility.

4.10.2 Formal definitions

The following formalism is based on algorithmic models of computing machines (Papadimitriou, 1994; Jones, 1997). The full formal analysis of the EVMA with asynchronous communication is not possible at this stage. This is mainly because of the current state of the theory of computation that is based on the Church-Turing thesis. Further investigations are required. Readers will also find interesting and fascinating implications discussed in the popular science book by Flake (2000). All the well-known properties of the computational complexity approach (Li and Vitányi, 1997) can be directly applied to the model presented here. This includes, for example, undecidability, the halting problem, and the concept of non-computable functions. (Note, we use the $\Sigma^*$ notation here, where $\Sigma$ means a set of terminals, and $\Sigma^*$ denotes a string comprising an arbitrary combination of characters from the set).

Definition 1. A virtual machine or a computing machine (or just a machine for short) is a tuple $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ where $K$ is the set of states and $s \in K$ is the initial state. $\Sigma_{in}$ and $\Sigma_{out}$ are sets of input and output symbols, respectively, referred to as input and output alphabets. $\delta$ is a function from $K \times \Sigma_{in}$ to $K \times \Sigma_{out}$, and is called the program. We say $\delta$ (or the program) runs on machine $M$. Remember that formally $\delta$ is an integral part of the machine itself. The notation $M(x)$ represents the output of machine $M$ given the input sequence $x$. $M(x, y)$ represents the output of machine $M$ given the input sequence $x$ followed by the input sequence $y$.

Definition 2. Suppose that $f$ is a function from $(\Sigma_{in})^*$ to $(\Sigma_{out})^*$, and let $M$ be a machine with input and output alphabets $\Sigma_{in}$ and $\Sigma_{out}$ respectively. We say that $M$ computes $f$ if for any string
\[ x \in (\Sigma_{in})^*, M(x) = f(x) \]. If such machine \( M \) exists, \( f \) is called a \textit{recursive function}. We also say that function \( f \) is computed by machine \( M \).

**Definition 3.** If for machine \( M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s) \) there exists a machine \( M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s') \) which computes \( \delta \), we call machine \( M \) a \textit{recursive virtual machine} or \textit{recursive machine} for short. We also say that function \( \delta' \) is an \textit{interpreter} of \( M \), and we say an \( M \) interpreter runs on machine \( M' \). We have \( \forall x \in (\Sigma_{in})^*, M(x) = M'(\delta, x) \).

**Definition 4.** Suppose we have a machine \( M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s) \) and there exists machine \( M_c = (K_c, \Sigma_{in_c}, \Sigma_{out_c}, \delta_c, s_c) \), where \( \Sigma_{in} \subseteq \Sigma_{in_c} \) and machine \( M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s') \) where \( \Sigma_{out_c} \subseteq \Sigma'_{in} \) and \( \Sigma_{out} \subseteq \Sigma'_{out} \). If \( \forall x \in (\Sigma_{in})^*, M(x) = M'(M_c(x)) \) then we say that \( \delta_c \) is an \textit{M compiler}, and we say \( M_c \) compiles \( M \) into \( M' \).

The emphasis in the conceptual framework presented above is to treat algorithms and running programs as \textit{machines} (recursive virtual machines to be precise). This, along with the notions of compilers and interpreters is discussed in length by Jones (1997). The above definitions do not make any assumptions about the number of states a given machine can have, nor about the storage capability. All possible models of computations, and different computer architectures (and Instruction Set Architectures, ISAs), fit the above definitions. For example one could use \( \Sigma \subseteq \text{Real} \) to perform analogue computation on real values. It can be shown that the proposed conceptual framework is a simple extension of the theoretical models of computation such as TMs and UTMs (Hopcroft and Ullman, 1979; Papadimitriou, 1994).

**Definition 5.** Let machine \( M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s) \), with finite input and output alphabets \( \Sigma = \Sigma_{in} = \Sigma_{out} \), \( \{\sqcup, \sqcap\} \in \Sigma \) and \( \{h, y, n\} \in K \). In other words the alphabet contains two special symbols, the blank and the first symbol, and there are three special state symbols, namely: \( h \) the halting state, \( y \) the accepting state, and \( n \) the rejecting state. We define three additional symbols, representing cursor directions: \( \leftarrow \) for “left”, \( \rightarrow \) for “right” and \( - \) for “stay”. If \( \delta \) is a function from \( K \times \Sigma \) to \( K' \times \Sigma \), where \( K' = K \times \{\leftarrow, \rightarrow, -\} \) then we say that machine \( M \) is a \textit{TM}.

### 4.10.3 Modelling hierarchies

The modelling methodology proposed here is based on the notion of recursive virtual machines. Due to the principle of universality of computation, all virtual machines are recursive. Also, based on the principle of universality, for any given machine \( M \) there exists an interpreter of \( M \) and compiler of \( M \) to \( M' \). It all collapses at the conceptual level to the universal virtual machine (UVM). Note however, that the universal virtual machine is not the main focus here. Rather it is the concept of machines that are domain-specific, constrained, and resource-limited that is the main application focus of the model.

We can model artificial and naturally occurring phenomena as a chain of virtual machines. One possible perspective on artificial life or evolutionary systems is to focus on a \textit{tower} of compilers.
and/or interpreters (see Figure 4.4). The concepts of chaining and stacking compilers and interpreters is discussed in detail by Jones (1997). The other approach is to use more traditional functional decomposition. All computing programs, including all evolutionary computation models, can be represented as a chain of compilers and/or interpreters, with different functional partitioning on each level. How this chain is constructed and how all its elements interact with each other is the issue that we are addressing about hierarchies.

4.10.4 Vertical and horizontal decomposition

Following the formal definitions, a machine can be statically represented as a program string, consisting of a prefix together with some instructions following this prefix. The prefix itself can be decomposed into another prefix and another program, and so on. This is called vertical decomposition, or a vertical hierarchy. Another type of decomposition is based on dividing a given machine into interacting parts – this is called a horizontal decomposition. Formally, a vertical hierarchy is based

\[
\begin{array}{cccc}
\text{In} & \text{Out} \\
\downarrow & \downarrow \\
M_0 & M_1 & M_2 & \text{Out} \\
\end{array}
\]

Figure 4.4: A vertical split of machine \( M \) into a tower of machines \( M_0, M_1, M_2 \).

on stacking interpreters and/or compilers (Jones, 1997), see Figure 4.4. A horizontal decomposition is based on splitting a single machine into two or more machines, see Figure 4.5.

Simple examples of vertical hierarchies include all sorts of (real-life) interpreters and compilers. For example given a Pascal interpreter written in Java we would have: program written in Pascal \( \rightarrow \) Pascal virtual machine (written in Java) \( \rightarrow \) Java virtual machine (written for example in C) \( \rightarrow \) C virtual machine \( \rightarrow \ldots \), where the arrow reads as “runs on” as encapsulated in Definition 3.

An example of horizontal partitioning would be functional partitioning of a single individual virtual machine. Let us imagine that we have a machine that can compute two operations on the natural numbers domain: addition and multiplication. If we perform functional partitioning, we can end up with two virtual machines, each computing a single operation, multiplication or addition, respectively. The union of these two gives us the original single machine.
One can enumerate through all the machine levels, starting from the base (fundamental) level $M_0$, up to the final highest-level machine, $M_n$. The actual input (instructions) are fed to the machine $M_n$.

It is important to remember that there is actually no special computational distinction between the *program* running on a virtual machine and the program emulating a particular machine itself.

All the interacting virtual machines are connected by their input/output streams. The hierarchical structure of that composition can have different forms, depending on the particular phenomena at hand. It can be a simple linear structure or it can be a tree-like structure. In general it is a directed graph, with cycles, self-referencing nodes, and possibly complicated interdependencies (see Figure 4.6).

![Figure 4.5: A horizontal split of machine M into machines M0, M1, M2.](image)
4.10.5 Partial equivalence

Some machines can be fully or partially equivalent to others, for example a Pascal virtual machine written in C and a second one written in Java are usually perfectly and fully functionally equivalent Pascal virtual machines, even though they use completely different machines at the lower level. Note that even though these two Pascal virtual machines have different machines below them, they can have exactly the same virtual machine one level down, for example a virtual machine for a particular operating system.

One can have a partial Pascal virtual machine that accepts a subset of all possible programs.
generated in Pascal. This is referred to as specialisation. On the other hand it is also possible to have a Pascal virtual machine accepting a subset of expressions from the C language, in addition to normal Pascal programs. The process of adding features to the language and enhancing the input language for a given machine is called a conservative extension (Jones, 1997).

Some machines can be recursively executed on themselves, for example a Java virtual machine interpreter written in Java and executed on a Java virtual machine interpreter. Some machines can be functionally equivalent, even though they use completely different language syntaxes or alphabets, for example expressions in prefix, postfix or reverse Polish notations. All these properties are well known in computer science, in which specific languages, interpreters and compilers flourish.

Suppose the problem at hand is coded in such a way that the solution can be expressed as a string of symbols from some language, $L$. For some languages, finding a solution (a particular sequence of instructions) is easier than for others. Coding of the problem is the key issue in solving the problem. In a sense the language $L$ captures and exploits some of the properties of the problem. This is one of the principle concerns of the proposed approach. With recursive virtual machines we have the necessary framework to model the transformations of a problem representation from one language to another, and we are able to translate the original problem into a more easily solvable equivalent.

4.10.6 Decomposition limits

Any particular level from the algorithmic hierarchy can be treated as a virtual machine that provides some functionality to the level immediately above it, and uses the level below to have the computation performed. In other words, a particular machine accepts input from one level, uses other levels to perform computation, and then passes the results to yet another adjacent level. The highest level of the chain of machines accepts some input (instructions), interacts with the level below it by sending/receiving some input/output, and returns some outputs (results) back. Similar to the base level, what we consider to be the highest level is also arbitrary. There is always a virtual machine feeding the instructions and accepting the results (e.g., a computer program or a human operator).

A given machine in a chain is formally equivalent to an interpreter or compiler of another machine located above it. The first (the base level) is the first interpreter, which we assume as being executed on some UVM. In the case of digital computers (and for the sake of simplicity) we can without loss of generality assume that the base level machine is equivalent to the UTM (Hopcroft and Ullman, 1979). Of course, this is an arbitrary choice, and the decomposition could be carried further, treating the UVM itself as a virtual machine, running on some software/hardware platform and so on, all the way down to electrical and/or chemical reactions and some physical processes\textsuperscript{13}.

\textsuperscript{13}Actually, according to Zuse (1970a) and Fredkin (1992) we have no reason to stop there, and we can decompose the system further, based on the idea that physical phenomena itself are running on some (digital, in the case of Fredkin’s theory) virtual machine.
4.11 Formal EVMA extensions

4.11.1 Equivalence classes

Let $\Sigma^B$ denote the set of all finite sequences of bits over binary set $\Sigma = \Sigma_{in} = \Sigma_{out} = \{0, 1\}$, and $\Sigma^B^*$ a set of all infinite sequences of bits each of which is a function that maps $N$ into $\Sigma$, where $N = \{0, 1, 2, \ldots\}$ is the set of natural numbers. Let $K^*$ denote the state domain of the computing machine, such that $K^* = \Sigma^B^* \cup \Sigma^B$. $K \in K^*$ can be used to denote a state subset, and $k \in K$ denotes a particular state of a computing machine. Accordingly, let $\Delta^*$ denote the set of all partial functions with a domain and co-domain in $K^*$, $\Delta$ denotes the set of all partial functions with domain and co-domain in $K$, and $\delta \in \Delta$ a specific program that maps $K \times \Sigma$ into $K \times \Sigma$.

A single synchronised sequential machine $M$ operating on binary sequences starts with state $s \in K$ and consists of a state set $K$ and mapping (program) $\delta$. Such a machine can run indefinitely (i.e. it can be a non-halting machine) or it can halt starting from a particular initial state $s \in K$ with the final state $k_f \in K$. In the traditional theory of computation, if two machines $M_1$ and $M_2$ start from the same state $s$ and halt with the same state $k_f$, we say that the machines are equivalent, $M_1 = M_2$. In this theory, non-halting machines belong to the same equivalence class.

The notion of equivalence can be used to provide useful formalisms and properties of traditional algorithmic computation, and the contemporary theoretical computer science is based on it. One of the fundamental properties of such a computational model is the fact that the program $\delta$ can be decomposed into individual steps, and chained together, forming a deterministic composition of deterministic components. Let $\delta_0, \delta_1, \delta_2, \ldots$ denote the decomposition of program $\delta$ into individual steps, called instructions. Let $s_0, s_1, s_2, \ldots$ denote the consecutive states of the computing machine, where $s_0$ is the initial state, and:

$$s_{i+1} = \delta_i(s_i)$$

(4.1)

This equation 4.1 emphasises the property of deterministic composition of deterministic components.

4.11.2 Beyond algorithmic function composition

The formalisms described above are based on traditional notions of computation and as such are also valid for EVMA. The traditional computational models as devised by Turing and others can, in simple terms, be thought of as sequential function composition in a strict mathematical sense. The EVMA however departs from, and generalises, the computational model presented above (Section 4.11.1). The generalised computational model of EVMA introduces two additional elements: a) the focus is on computation that never halts (non-halting programs); and b) the computation of a single virtual machine can be potentially based on a collection of asynchronously communicating machines. Given these two elements, the properties of traditional computational models of sequential computation are
of limited practical applicability to the EVMA model. Similarly with respect to differences between single synchronised deterministic computation of traditional TM models and multi-threaded computation (as discussed in detail e.g. by Lee (2006)), there are significant differences between the EVMA and the simple computational model based on algorithmic computation. We will discuss these below, starting from the concept of an equivalence class, and following on into issues of deterministic decomposition.

**Identity and equivalence classes.** Traditional model computations that do not halt belong to the same equivalence class which, in the context of EVMA, is not sufficient. We want to be able to differentiate between various non-halting computations. Some of them might be considered useful and should be carried on, while some may be considered useless and should be forcefully terminated by the environment. Unlike the traditional algorithmic computation equivalence class problem, where we have a clear formalism to follow, the issues related to equivalence within the EVMA are basically uncomputable in the algorithmic computation sense. The problem of equivalence is unsolvable within traditional algorithmic computation. In the EVMA this issue is left unspecified. In the EVM, we have adapted a generic approximation technique based on a particular variation of a tagging mechanism. Each computational cell constructs a particular hashing function $PE$ (initially an identity function) takes the initial stack and the program itself as an argument. $PE$ is computationally bound – which means that it is guaranteed to halt within a specified limit of computational resources. $PE$ produces a sequence of integers. Two programs are considered pseudo-equivalent if their respective $PE$ function provides the same result.

$$PE(stack_0, program_1) = PE(stack_0, program_2) \quad (4.2)$$

Note that $PE$ is not fixed, but it is subject to modification during the cell’s life cycle. Each computational cell is seeded with the initial identity-based $PE$ function.

**Non-determinism and asynchronous computational composition.** As discussed earlier, machine $M$ can be composed of a number of concurrently and asynchronously processing programs $\delta_i$. The property of deterministic composition of deterministic components from Equation 4.1 does not hold anymore. Consider for example two programs $\delta^1$ and $\delta^2$ executing concurrently within the scope of the same machine $M$:

$$s_{i+1} = \delta^k_i(s_i), \text{ where } k \in 1, 2 \quad (4.3)$$

At each step $i$ either $\delta^1$ or $\delta^2$ can provide the next state of the machine $M$. There are two cases: a) the interleavings between these two programs is itself a computable function, in which case the state space could potentially be extended to provide the cross-product of all the possible interleavings, and the model would fall back into the traditional deterministic model discussed earlier; or b) the
Interleavings between the programs is uncomputable, in which case no formal analysis would be appropriate. For many practical applications to date, notion (a) has been assumed. However, in principle the EVMA assumes notion (b). For example, consider a multi-cell EVMA environment where all the cells are seeded with a random programs that depend on the execution of other programs. Some of the initial programs will terminate, some will not. Due to the fact that some of the dependencies may have not be computed (halting problem), the state of the collection of cells is in itself uncomputable in the TM sense. If the execution of only some of the programs is dependent on the state of execution of other programs, the overall computation conducted by a single cell requires the knowledge of the current state of all other executing cells, including the ones that have not halted yet, or, may never halt.

To address some of the practical limitations of this general model of computation, another approximation technique is employed by the EVMI. We use the notion of the Solomonoff universal prior (Hutter, 2002) and the approximation of Chaitin’s halting probability $\Omega$ (Chaitin, 1999), to approximate the state of the executing cells.

4.11.3 Generalised EVMA computation

To express a computation performed by an executing EVMA system, imagine that EVMA maps an evolving bit pattern (or in the current EVMI, an integer sequence) into an evolving bit pattern (integer sequence).

If we denote a partially or totally ordered set of tags as $T$ and EVMA computation as the mapping $E$, then formally the computation of EVMA can be expressed as:

$$E : (T \rightarrow K) \rightarrow (T \rightarrow K)$$  \hspace{1cm} (4.4)

This formulation is not unique to EVMA itself, and has been used to model concurrent processing and process algebra. Good introduction has been provided by Lee (2006).

4.12 Summary

In this chapter we have provided an introduction to the contemporary theory of computation, discussed various aspects of the contemporary theories, and presented our own extensions to the theory of computation based on EVMA. We have provided details of the existing algorithmic theories, shown some examples of other formulations, and argued for the need for formalisms that can describe a large number of autonomous asynchronously communicating entities. We have introduced the notion of a computing machine, discussed the physical limits and expressiveness of various computational models, and briefly introduced the concepts related to hypercomputation. We then followed this with a discussion on hierarchical computation and provided a formal model of hierarchical computation built on top of traditional algorithmic models.
We have then extended the traditional algorithmic computation model by our own EVMA computation model, and provided more detailed formulations and extensions enabled by the EVMA. We have also discussed the differences between EVMA and EVMI in terms of the currently implemented set of features.

In the following chapter we will analyse and discuss the notion of EVMA mapping of evolving bit patterns in the context of artificial and biological evolutionary systems. Firstly, we will discuss various theories of evolution from a biological perspective (next chapter) followed by a discussion of artificial evolutionary systems (Chapter 6). We will draw parallels from these various evolutionary models and our EVMA computational architecture.
Chapter 5

Theory of Evolution
The human brain became large by natural selection (who knows why, but presumably for good cause). Yet surely most “things” now done by our brains, and essential both to our cultures and to our very survival, are epiphenomena of the computing power of this machine, not genetically grounded Darwinian entities created specifically by natural selection for their current function.

(Gould, 1987a)

5.1 Theory of Evolution

5.1.1 Answers that we seek

The theory of evolution is often portrayed as a unified, consistent and sound model that was proposed by Charles Darwin and that is confirmed and expanded by an extensive body of empirical studies. This view is widespread in scientific circles, popular science magazines and mass media. This is however a somewhat naive and simplified view of evolution. The theory of evolution is complex and inaccessible to most people outside of biology research. We will try to present a broader picture of the contemporary theories of evolution: their genesis, assumptions and implications. The simple Darwinian evolutionary model does not account for all of the observed phenomena, and it was never intended as such an explanation. Further studies and explorations are needed. Especially in the realms of highly complex biological structures, such as human societies, there is evidence that Darwinian principles must give precedence to other mechanisms and factors. In this chapter, we will review the rich and complex landscape of contemporary theories of evolution. This is to provide enough background understanding of complexities of biological evolution, before we proceed into the discussion of artificial evolutionary models and computational evolution.

The simple Darwinian model is widespread in contemporary biologically-inspired computational architectures. In particular, it is used as a founding assumption in many evolutionary computation models. Genetic Algorithms, Genetic Programming and other related models of evolutionary computation use a simplified notion of Darwinism. However, the biological theory of evolution as it is understood today is complex, diverse and, in some aspects, substantially departs from the original formulation of Charles Darwin. It is for historical reasons and due to Darwin’s influence and extensive studies, that the contemporary evolutionary theories use an umbrella term Darwinism. Charles Darwin is attributed with the founding role in the development of the theory.

In this section we present the scope and the contemporary views on the theory of biological evolution. We present how diverse and complex the body of research is, and how many individual theories comprise the umbrella term “theory of evolution”. We also put the original contribution of Charles Darwin in its historical context. The aim of this section is to provide an overview and better

\footnote{Note, the first formulation of the concept of natural selection is attributed to, among others, Charles Darwin’s grandfather, Erasmus Darwin (King-Hele, 1999).}
understanding of contemporary evolutionary biology and the background of the current models of biological evolution.

The contemporary theories of biological evolution depart from naive models that are used in *evolutionary computation* (EC). One of the goals of the EVM architecture is to bridge the gap between the contemporary theories of biological evolution and models used in evolutionary computation. The main aim here was to use biological inspirations in the design of the EVMA. We discuss the various biological elements that are reflected in the EVM model. The EVM architecture departs substantially from the simplistic Darwinian models, and rather takes its inspirations from the advances in contemporary genetics and theoretical evolutionary biology. Although used exclusively in the realms of computing systems, the EVMA provides insights and helps us to better understand evolution in biological systems. Some properties of the EVM model are, in fact, already coincidentally the subject of speculations and active research efforts in genetics and theoretical evolutionary biology. Among other features these include: exaptation, momics, epigenetics and symbiogenetics (these mechanisms will be discussed later in this chapter).

We propose the basic mechanisms of selection subjected to evolutionary pressures that work at the same time on different units of heredity. This process is abstracted into a meta-level evolutionary process, where complex units are composed out of interactions of simpler lower-level units. We test this model in the artificial computational evolutionary system. Whether this new model of evolution and heredity is applicable in the realms of biological world is not investigated. Investigating this relationship is beyond the scope of this thesis. The objective for this research is to prepare and test the EVM model only in computational settings. The EVM prototype makes possible further studies within artificial life, computational evolution and biological systems. Comparisons and studies of biological evolution in the context of the EVM model are planned as further work.

We start with a historical account of pre-Darwinian formulations. After that we follow to the original Darwinian formulation and summarise all the theories that comprise the modern synthesis of evolutionary theory. This will provide background information, and it will establish how the EVM architecture parallels contemporary formulations of the theory of evolution and the current understanding in evolutionary biology.

### 5.1.2 Pre-Darwinian formulations

We do not intend here to provide a full review of different historical aspects of the origins of the theory of evolution. However, our goal is to present the complexities together with the diversity of the theory itself. We want to stress some common misconceptions related to the theory of biological evolution. The best way to achieve that is to provide a historical analysis on the origins of the different theories and present their inter-dependencies. From the contemporary writings in popular magazines and in some scientific publications it may appear that the theory of evolution is a simple single coherent
theory (or even a simple fact). That view is misleading\(^2\).

The general conception of individual organisms and their clustering into species have been present before any of the mechanisms of speciation have been proposed. Thus, the general framework and conceptualisation discussing biological evolution was established and present before the basic mechanisms of natural selection and random mutations\(^3\). Charles Darwin formulated the basic notions of biological evolution based on two combined concepts: selection, and random mutations. He did not know, however, the exact mechanisms of how hereditary information is passed from generation to generation. Jean-Baptiste Lamarck proposed some mechanisms, and explained how propagation of traits might occur. Lamarck proposed that acquired skills and features might be passed on from generation to generation, directly. Later, Mendel was the first to propose the precise mechanisms that have been later confirmed by the discovery of the double helix structure\(^4\).

Many historians and biologists have investigated the history and development of evolutionary theory, and have asked questions related to the origins of the theory of evolution by means of natural selection, e.g. King-Hele (1999) and Bowler (1989).

By explaining the historical context of the theory of evolution, our intention is to clarify some of the difficulties and questions that different aspects of the theory of evolution (as in the modern evolutionary synthesis and beyond) have been and are trying to address. Each individual aspect of the modern theory is a self-contained scientific theory. These various theories address different data, and use different evidence to support postulated models. Some of these theories were postulated early by evolutionary biologists but were not picked up into the current mainstream theories. Some were proposed recently, long after the original Darwinian formulation, and some are being postulated by contemporary biologists now, based on recent advances in genetics, theoretical biology, and other

\(^2\)In the context of recent discussions between evolutionists and creationists, especially in the USA, it is important to understand that the evolution of species is a well established and accepted scientific model. However, the exact mechanisms and all the details of the process are still to be explained. The basic model is continuously being extended and new details are provided to explain various effects and phenomena.

\(^3\)The common observation of biological organisation divided into individual organisms, clustered into species, skewed and locked the overall approach to biological evolutionary theories. This might be also the reason why theories such as Eigen’s hypercycles with quasi-species, and Maturana-Varela’s autopoiesis, struggled initially to be picked up by the mainstream, due to their departure from the fundamental simplistic grouping of biological units. Recent discoveries change this view. We witness a shift in the general understanding of the basic mechanisms of biology, together with a more holistic approach to biological evolution. That shift is analogous to the one in physics: from the classical to the relativistic theory of the basic mechanisms. From that perspective, this is a profound shift. From initial vitalistic and almost mystical or spiritual theories, through a purely reductionist approach, back to holistic and relativistic theories. Biology, as such, seems to be closing a full circle in the scientific efforts. More biologists realise that the reductionist approach alone may not be the best approach in understanding the complexities of biological evolution. These views do not fundamentally conflict with the original Darwinian formulation, but rather expand it and provide a more detailed and accurate picture of the mechanisms of biological evolution.

\(^4\)It was effectively Mendel who proposed the basic mechanisms that contemporary Darwinism is based on. A more appropriate name for this body of research would be Mendelism – this name is used instead of Darwinism in some biological circles.
related fields.

There is compelling evidence that the precursors to the Darwinian formulation of evolutionary theory have existed for over 100 years before the actual formulation by Wallace and Darwin (Glass, Temkin, and Strauss, 1968). One of the first names associated with the idea of natural selection is Pierre Louis Moreau de Maupertuis. A French mathematician and astronomer, de Maupertuis as early as 1745 formulated ideas in biology that are considered significant precursors to the later developments of theory of natural selection. It is sometimes concluded that De Maupertuis contributed the original account of the theory of natural selection. He wrote in 1745, over 100 years before Charles Darwin:

Could one not say that, in the fortuitous combinations of the productions of nature, as there must be some characterised by a certain relation of fitness which are able to subsist, it is not to be wondered at that this fitness is present in all the species that are currently in existence? Chance, one would say, produced an innumerable multitude of individuals; a small number found themselves constructed in such a manner that the parts of the animal were able to satisfy its needs; in another infinitely greater number, there was neither fitness nor order: all of these latter have perished. Animals lacking a mouth could not live; others lacking reproductive organs could not perpetuate themselves. The species we see today are but the smallest part of what blind destiny has produced...

(de Maupertuis, 1745)

Another important figure here is Erasmus Darwin (the grandfather of Charles Darwin). Erasmus’s most important scientific work is the publication of Zoönomia (1794–1796). In Zoönomia, Erasmus anticipated some of the ideas of chance variation and natural selection. Zoönomia is widely considered to foreshadow the pre-Darwinian theories of Jean-Baptiste Lamarck, and the theory of evolution formulated by his grandson, Charles Darwin (King-Hele, 1999). The following passage from Zoönomia explains that the same kind of living filament is and has been the cause of all organic life:

Would it be too bold to imagine that, in the great length of time since the earth began to exist, perhaps millions of ages before the commencement of the history of mankind [...] all warm-blooded animals have arisen from one living filament, which the great First Cause endured with animality, with the power of acquiring new parts, attended with new propensities, directed by irritations, sensations, volitions and associations, and thus possessing the faculty of continuing to improve by its own inherent activity, and of delivering down these improvements by generation to its posterity, world without end!

One of the first formulations of the principle of natural selection that is officially acknowledged is the work of William Charles Wells (1757–1817). Wells’s paper was read before the Royal Society and published in 1818. It included two essays in which Wells had assumed that there had been evolution
of humans and recognised the principle of natural selection. Charles Darwin in the fourth edition of his “The Origin of Species” (Darwin, 1859) acknowledged:

In this paper he [Wells] distinctly recognised the principle of natural selection, and this is the first recognition which has been indicated.

Another important work contributing to the formulations of Darwinism is the work of Herbert Spencer. Spencer, after reading Darwin’s work, coined the term *survival of the fittest* and proposed a basic formulation of the evolutionary theory based on natural selection in “First Principles” (originally published in 1862) (Spencer, 2002, 6th Edition). In this work Spencer sketched the general properties of the theory of evolution, which he applied to biology, sociology and the study of the physical evolution of the world. He maintained that all the phenomena manifestations proceed from their source according to a process of evolution. Herbert is considered an influential figure who helped to encourage the scientific community to accept the Darwinian theory of evolution.

### 5.1.3 Darwinism

There is some confusion in popular science magazines and in literature regarding the term *Darwinism*. Some definitions use the term *Darwinism* to refer exclusively to the original formulation of Charles Darwin alone. However, the actual formulation of Charles Darwin is often misunderstood. It is complex and difficult to express as a single coherent theory. It is open to different interpretations that may lead to speculation. In this context it is also worth remembering that the work of Charles Darwin builds on and derives from work of his predecessors.

Sometimes the term *Darwinism* is used to mean the most recent synthesis of all mainstream evolutionary theories whereas some use that term to denote any theory that derives from the notion of natural selection in general. That is confusing, as it overshadows important contributions by post-Darwin researchers. To provide some clarification of the issue, in this work we will use the term *Darwinism* exclusively to refer to the original contribution of Charles Darwin himself. We will refer to the overall modern synthesis of the theory of evolution (in the broader concept of Darwinism) as the modern synthesis (or neo-Darwinism). And we will refer to the overall body of contemporary theories of evolution in general, including different, somewhat speculative contemporary theories, just simply as the *theory of evolution*. The theory of evolution includes the body of work contained in the modern synthesis, and the somewhat out-of-the-mainstream theories, e.g. symbiogenetics, epigenetics, rnomics, exaptation, etc.

The theory of evolution, as we understand it today, is a collection of many contributing theories. Hence, in this context the term "theory" of evolution is just an umbrella term – a figure of speech – that departs from its strict scientific notion. Some of those theories are accepted by mainstream evolutionary biologists and are referred to as the modern synthesis (or neo-Darwinism). It is worth noting that all those theories have been developed over a long period of time, prior to Darwin, by
Darwin himself, and by many contemporary evolutionary biologists and researchers from different fields. All those contributing theories are complementary scientific efforts and results, all geared towards the ultimate understanding of the biological evolution.

**Darwinism – the theory.** Charles Darwin formulated his theory in terms of the twin concepts: chance of variation and natural selection (Darwin, 1859). Darwin shared with other researchers of his times the assumption that the biological characteristics of an individual represent somewhat those of its parents. He also assumed that both parents contribute more or less equal parts to this mixture in their offspring. This means that an offspring of a parent with a useful chance variation in its characteristics would inherit on average 50 per cent of these new characteristics. Note, that this offspring would be able to pass on average only 25 per cent of them to the next generation. Thus the new characteristics would be diluted rapidly, with little chance of establishing themselves through the process of natural selection. This means that the original formulation proposed by Darwin (that we refer to here as Darwinism) could not explain the emergence of new characteristics in the evolution of species.

The Darwinian theory of evolution introduced a radically new understanding of the origin and transformation of species. However, it could not explain the persistence of newly evolved traits. Darwin himself recognised the inability of his theory to explain the constancy of hereditary traits. It was a serious flaw for which Darwin himself had no remedy.

At the same time, independently from Darwin, another British naturalist and biologist, Alfred Russel Wallace, was collecting data and specimens in the Amazon rain forest in Brazil. Wallace’s objectives were aligned with those of Darwin – the main motivation was to solve the problem of the origin of species. Wallace was one of numerous correspondents from around the world whose observations Charles Darwin used to support his own theory. In 1858, Wallace wrote an essay (Wallace, 1858) that prompted Darwin to publish his own results earlier than intended. The Wallace and Darwin formulations are considered to be equivalent in a sense that both postulated the notion of natural selection and chance variation.

### 5.1.4 Modern synthesis, neo-Darwinism

A few years after the publication of the Wallace and Darwin work (Darwin, 1859), there were a series of important breakthroughs. First, in 1863, Gregor Mendel, based on his experiments with garden peas (Tudge, 2002), deduced that there exist special units of heredity. Mendel referred to them as traits. He postulated, that these units do not blend in the process of reproduction, but are transmitted from generation to generation without changing their identity. This was a crucial discovery. It solved the problems with the original Darwinian formulation. It was not popular during Mendel’s life and the scientific community (including Charles Darwin himself) used a different notion of hered-

---

5There is an established priority granted to Darwin, because of his unpublished paper from 1844. However, the Wallace role and input into the field and the final formulation should not be underestimated.
ity (based on pangenes) instead of Mendelian genetics. With this discovery it could be assumed that random mutations, or as Charles Darwin referred to them, chance variations, will not disappear within a few generations, but will be preserved and can be potentially reinforced by natural selection. Unfortunately, this important discovery was not widely spread in the scientific community and the implications of Mendel’s discovery were not known in the mainstream of the scientific endeavours for several decades. The basic mechanisms were rediscovered at the beginning of the twentieth century by Bateson (1902).

In 1910, Thomas Hunt Morgan discovered a special case of a mutant fly. He found that this condition was inherited precisely as a Mendelian recessive trait (Garland, 1978). This provided a link between experimental biology and emerging evolutionary theory, combining Mendelian genetics and natural selection. In the subsequent years, Morgan and his colleagues developed the Mendelian-Chromosome theory of inheritance. Morgan published “The Mechanism of Mendelian Inheritance” in 1915. By that time, most biologists accepted that genes situated linearly on chromosomes were the primary mechanism of inheritance.

Another important event in the formulation of the modern synthesis was the work of Ronald Fisher (1911) and the publication of the foundational article, “The Correlation Between Relatives on the Supposition of Mendelian Inheritance” (Fisher, 1918). Fisher proposed and investigated a genetic model arguing that a continuous variation amongst different characteristics can be a result of the Mendelian model of inheritance. Fisher, using his model, showed how continuous variation could be the result of the action at many discrete loci. Fisher’s argument is regarded as a main, founding point of the modern evolutionary synthesis: neo-Darwinism.

According to the modern synthesis, genetic variation in populations arises by chance through mutation and recombination, i.e. crossing over of homologous chromosomes during meiosis. Evolution consists primarily of changes in the frequencies of alleles between one generation and another as a result of (Wright, 1931; Haldane, 1932; Huxley, 1942; Mayr and Provine, 1980):

- genetic drift,
- gene flow, and
- natural selection.

The modern evolutionary synthesis continues to be developed and refined. The most notable progression was a shift from a species-centric and individuals-centric view to the orthodox gene-centric view of evolution. The orthodox gene-centric view persists as the mainstream in current biology. Originally proposed by George C. Williams in the 1960s, the gene-centric view of evolution, or gene selection theory, holds that natural selection acts at the level of the gene. This is based on the assumption that the genes which by phenotypic effects successfully promote their own replication will

---

6 An allele is a set of viable DNA codings of the same gene, occupying a given locus on a chromosome. In general, an allele is a set of possible values for a position of a generic sequence.
increase in frequency (Williams, 1966). Other researchers, including W. D. Hamilton, John Maynard Smith, and Robert Trivers, contributed to the shift in biological thinking at this time. Richard Dawkins popularised these views by his 1976 book “The Selfish Gene” (Dawkins, 1976). This work can be seen as having been built upon the work of Fisher (1930). In the extreme version, maintained by some popular scientific magazines, the orthodox gene-centric view maintains that each gene (or set thereof) is associated with a single specific trait. This simplified view of evolutionary processes has been proved insufficient, and it has been refined and reformulated by recent advances in evolutionary biology.

### 5.2 Neo-Darwinism

#### 5.2.1 DNA and RNA

The existence of *deoxyribonucleic acid* DNA, (Figure 5.1), and *ribonucleic acid* RNA was discovered in the middle of the 19th century. Though many earlier discoveries and research started suggesting that DNA might store genetic information, it was not until the 1953 *Nature* publication (Watson and Crick, 1953) that the details of the structure of the DNA molecule were proposed and the process of protein production from nucleic DNA were described. It has been known since that DNA is a nucleic acid that contains the hereditary information of all cellular forms of biological life on Earth.

![Figure 5.1: Space filling model of a section of a DNA molecule.](image)

135
A strand of DNA is divided into triplets, that are called codons. Codons contain genes, areas that regulate genes and areas of (yet) unknown function. DNA is organised as two complementary strands. It is composed from four bases, identified by letters: A, T/U, C, G\(^7\). A base on one strand of DNA will connect properly only with one base on the other strand: T with A, A with T, G with C, and C with G. In most biological organisms DNA is transcribed into ribonucleic acid molecule (RNA), i.e. the cell makes a negative copy of DNA into a messenger RNA (mRNA). A negative copy, single-stranded RNA molecule of a gene which is used to decode the original gene by specifying the amino acid sequence of one or more polypeptide\(^8\) chains. See Figure 5.2 for a schematic view of the chemical structure of DNA. The mRNA passes out of the nucleus and into the cytoplasm. In the cytoplasm it is latched onto by decoding particles, called ribosomes\(^9\). Associated with the ribosome in making proteins are the transfer RNA molecules (tRNA). These tRNAs bring the activated amino acids, covalently linked to one end of their structures, and recognise the codons in the mRNA using anticodons at their other ends (Carter, 2005).

---

\(^7\)Adenine, Thymine/Uracil, Cytosine, Guanine

\(^8\)Peptides are macromolecules consisting of several amino acids linked via amide bonds between their \(\text{COOH}\) and \(\text{NH}_2\) groups. Peptides do not necessarily adopt a defined spatial structure, unlike proteins. The difference between polypeptides and proteins is not yet fully known.

\(^9\)Ribosome – particle composed of ribosomal RNAs and proteins. They catalyse translation of mRNA to protein.
5.2.2 Exons and introns

RNA is divided into regions called exons. Exons are protein coding sequences of the primary RNA transcript. During the first decade of molecular biology in the 1960s, it was believed that the transcript is composed exclusively of exons. However, there was an important discovery that in the primary transcript neighbouring exons are separated by non-coding regions called introns. An intron is a segment of DNA that is transcribed into mRNA but is then cut off from the RNA, leaving behind the exon sequences to be translated into polypeptides.

To form the mRNA that effectively codes for proteins, the introns are removed, and the exons are spliced together. In some organisms, the machinery for splicing exons is accomplished not by a protein catalyst or enzyme, but by an RNA catalyst, or ribozyme. Many of these ribozymes are actually themselves introns. Thus, in some cases, the elaborate machinery used for splicing exons is assembled on the introns themselves. It removes an intron at the right place and joins the neighbouring exons together. This must be done in the right direction and place. It makes a difference whether the exons are spliced by even one letter off, because it renders a completely different overall sequence. It is similar to a computer program – if only a single instruction is removed, the overall behaviour of the computer program may be completely different.

There is an interesting parallel between the structures of mRNA and tRNA: tRNA is made as a long transcript with several embedded tRNA genes that have to be cut out. This is analogous to exons and introns in mRNA strands. During the transcription all the information from the DNA is copied to mRNA.

5.2.3 Orthodox gene-centred view

The discoveries and efforts in molecular and evolutionary biology in the 20th century have led to a better understanding and, to a certain extent, provided an important background for the unification of evolutionary theories. The common understanding of Darwinism is somewhat different of the one presented above, and in most cases represents a simplified view of the actual state of the art within the biological community. The popular understanding of Darwinian evolution is based purely on the orthodox gene-centred view. This view, in its naive form, suggests that there is a clear functional mapping between the genetic information (the genotype) and the organism’s structural and functional properties (the phenotype). This popular misconception is widely evident in various fields, including the field of evolutionary computation. We will discuss the implications in Chapter 6. In this text, when we discuss the orthodox gene-centred view, we refer to the biological understanding of the theory, not to the popular, naive views, unless stated otherwise.


11Note that this portrays quite a complex network of dependencies within the process of coding and splicing exons. A detailed discussion on ribozymes is provided later in Section 5.3.1
5.2.4 The search for a unifying theory

Even though the advances and achievements of the modern synthesis provided a unification of many distinct theories and areas of research, there is still no single unifying theory that would provide an account for all the different cases observed in the biological world. There are some theories that have not been adapted into the mainstream, and not all the questions have yet been answered. While the basic mechanisms of genetics and inheritance are better understood, the questions related to prebiotic and pre-DNA evolution are even more exposed. These questions relate to units of heredity. How did they come into being? What process created those units (DNA molecule, RNA molecule, or other), so they can be mutated and selected (as postulated by neo-Darwinism)? These and other questions are being more recently re-evaluated and investigated. These questions are not fully answered by neo-Darwinism, hence the search for a more general and unifying theory of evolution continues.

In the following sections we will present some contemporary views from evolutionary biology and genetics. We do so for two reasons. First, we want to demonstrate that most of the contemporary evolutionary computation models are rooted in the old, more naive interpretation of genetics and evolutionary biology. These models are out-of-sync when compared to the state of affairs in contemporary biology. The second reason is to show parallels between the EVM model and theories from contemporary genetics. It is possible, although there is not enough evidence yet, that the overall interactions of different enzymes and catalysts together with different DNA and RNA molecules create a large, complex asynchronous and parallel machine. This machine could be equivalent to a giant EVM universe (discussed later). Firstly, this offers interesting modelling and computational parallels between our EVM architecture and the biological evolutionary systems. The second, more speculative, aspect is the possibility of hypercomputation being inherent in the asynchronous interactions of different elements and levels of the evolutionary system itself. We mean here hypercomputation mostly by means of trial-and-error machines (Bringsjord and Zenzen, 2003).

5.3 Beyond neo-Darwinism

Recent advances in molecular biology have seriously challenged the common understanding of genetic stability itself. The core of neo-Darwinism and the orthodox gene-centred interpretation of biological evolutionary theory is undergoing a major revision (Capra, 2002). The entire image of genes as causal agents of biological life – the image that is deeply embedded in both mainstream popular and scientific thought – continues to be under thorough investigation. There is also critical analysis that follows from inadequacy of theory of natural selection. Some authors argue that selection is ill-formulated principle, and should be replaced by other models, such as self-organisation (de Faria, 1988). Others argue hierarchical nature of the selection mechanisms (Keller, 1999; Okasha, 2006), and this is something that EVMA tries to accommodate. In this section we provide a brief account

---

12“Before life”, referring to the time before life appeared on the Earth.
of some of the areas of contemporary evolutionary biology that are in the forefront of the theoretical evolutionary biology. Our own computational model takes inspiration from the following theories and ideas: Rnomics, epigenetics, exaptation and symbiogenesis.

5.3.1 The RNA world

The RNA world hypothesis (Woese, 1967) proposes that RNA was the first life-form on earth, and that DNA was used later, purely for archival purposes. RNA molecules developed cell membranes around them and became the first prokaryotic cells. This hypothesis is supported by the RNA’s ability to store, transmit and duplicate genetic information. Because RNA can reproduce on its own, and it can also perform all the tasks of both DNA and proteins (enzymes), RNA is believed to have once been capable of independent life.

A ribozyme (ribonucleic acid enzyme, or RNA enzyme) is an RNA molecule that catalyses a chemical reaction. A ribosome is an organelle composed of ribosomal RNA and ribosomal proteins. It translates mRNA into a protein. Ribosome produces proteins by linking together amino acids according to the triplet code in the mRNA. It is like a factory that builds a protein from a set of genetic instructions. The most magnificent relic of the prebiotic evolution is the ribosome itself, which is likely just a ribozyme that has picked up accessory proteins to stabilise itself. Ribosome functions are all based on RNA, and there is considerable evidence now that proteins are unnecessary for the actual functions of the ribosome.

Eukaryotic nuclei contain a large number of different RNA, besides those RNA used to generate ribosomes and tRNAs. These RNAs are called heterogeneous nuclear RNAs (hnRNAs). Many of these control gene expression, and are active during the splicing reactions. Some non-coding RNAs called microRNAs (miRNAs) regulate the production of proteins coded in other genes. Researchers now believe that at least some introns have a crucial role in determining the behaviour of the actual genes, e.g. the order in which they are switched on (Mattick, 2001). There are numerous levels at which gene expression is controlled after transcription, and hence are entirely at the RNA level.

This suggests that the hereditary information is not exclusively stored in genes as such. It can be (and most likely is) stored in RNA and other auxiliary structures. The emerging picture of functional dependencies between different genetic structures is more complex than originally thought. It suggests that the orthodox gene-centred view is too simplified and not an accurate representation of the actual genetic processes.

5.3.2 Exaptation

One of the two main dogmas of the modern synthesis is the notion of the Blind Watchmaker (Dawkins, 1976). The second one is gradualism. Gradualism holds that biological evolution of a certain structure or function occurs through the accumulation of slight modifications of an original structure over a period of generations. An interesting aspect of modern evolutionary theories is the long-lasting
conflict between gradualists and proponents of alternative views on mechanisms of adaptability. The main opponents of traditional gradualism suggest that evolutionary change may happen in various different ways, and is not exclusively limited to gradualism.

Support for gradualism derives from the fact that many, probably most, evolutionary processes are indeed gradual and incremental. However, it is one thing to believe that gradual processes predominate in nature, and quite another to hold that all evolutionary processes must be gradual. The issue is, after all, simply an empirical one; even if no nongradual changes were ever witnessed, one could never exclude the possibility that the next evolutionary process to be uncovered might be nongradual.

(Gould and Vrba, 1982)

For example, a bird’s wing would not be of much use at a fraction of what is needed for flying. If there were no adaptive value to the earlier structures of a wing, it would seem unlikely that the trait would survive long enough to become useful to the organism.

Gould and Vrba, using earlier theories on pre-adaptation, coined the term *exaptation* to explain how such traits may evolve. According to exaptation theory, complex physical traits evolved from earlier traits that had a different adaptive value. The original adaptive value might have been completely unrelated to the final trait. For example, the wings of insects and of birds are thought to have evolved originally for a quite different function than flying. They evolved, most likely, for thermo-regulation. A fraction of a wing, for example covered in air-trapping feathers as in birds, could have been an efficient thermo-regulator. It could be used as a cover or a fan to be used when too cold or too warm. Organisms with such an adaptation would thus be fitter than those without, leading to the spread of this particular trait. The thermo-regulator would grow larger over time. Eventually, the thermo-regulator would have grown so large and at some point it would allow insects (and birds) to glide. At this point the structure would have become an entirely new trait. Organisms with wings could glide. This could greatly improve their mobility. Now, gradual evolution could take place on the wing as a flying trait, instead of the wing as a thermo-regulator. The structure of the wing would become entirely adapted to flying. Many similar arguments are provided by Gould in his work and writing, e.g. (Gould, 1992).

In the context of computational systems, exaptation would suggest that chunks of code, used in one context, can be re-used in completely different contexts, when these become available. In traditional *evolutionary computation* (EC) systems this is not possible. The code has a clearly defined context and purpose, and any bloat or not-currently-used code is treated as junk and pushed out of the gene pool. The so called *parsimony pressures* are used to remove any possibility of exaptation from EC systems. In this regard the EVM architecture differs substantially from Genetic Programming (GP) or other contemporary EC systems. There is no explicit parsimony pressure as in GP systems.

---

13 The assumption that a program should not contain any unnecessary pieces of code.
Chunks of code are utilised in different contexts. This works not only across multiple problem search spaces, but also across multiple levels in the same problem space. Some pieces of code (loops, or other related control structures) can be re-used on multiple levels of abstractions, in different contexts yielding different results, in each individual case, though, they are useful for the overall performance of a given bigger program. This modularisation of computation in multiple contexts and on multiple levels of abstractions is one of the contributions of our EVM architecture in the field of evolutionary computation. We will provide more detailed discussion of this topic in Chapter 6.

The architecture proposed and the results obtained through the study of the EVM architecture are similar to studies in biomusicalogy and the evolution of language. In these areas, studies suggest that syntax and language (considered to be inherently hierarchical structures) could not have been evolved in a gradual fashion; “the difference between flat structure (beads on a wire) and hierarchical is absolute” (Bickerton, 1999) page 159.

### 5.3.3 Symbiosis

The traditional evolutionary views stress the role of competition. This is due to the fact that most of the interactions between species involve competition for the same food supply. From that perspective, the interactions can be simplified to just two activities: eating (predation) and avoiding being eaten (avoiding predation). However, these interactions are often brief. There are many cases, where two (or more) species live in close association for long period of time. Such associations are called *symbiotic*, i.e. living together. Symbiosis is a term depicting a symbiotic relationship. The species involved in such relationships are called *symbionts*. In symbiosis, at least one member of the pair benefits from the relationship. We distinguish 3 different kinds of symbiosis, depending on how the other member is affected:

**Mutualism.** Symbiotic relationship in which each species benefits. Many mutualistic relationships become obligative: neither species can live without the other. For example, protozoans in the gut of termites digest the wood ingested by the termites.

**Commensalism.** A type of symbiosis where two (or more) organisms from different species live in close proximity to one another. In this type, one (or more) member(s) is unaffected by the relationship and the other members benefits from it. In commensalism, one species (the commensal) obtains nutrients, shelter, support, or locomotion from the host species, which is substantially unaffected. E.g., sharksuckers (remoras) obtain locomotion and food from sharks.

**Parasitism.** Symbiotic relationships in which one organism (*parasite*) lives on or in the body of another organism (*host*). From host’s tissues, the parasite gets its nourishment. The parasite does some damage to the host. Simplified, parasitism is a relationship between two species in which one benefits at the expense of the other.
There are different categories of parasites. The main classification distinguishes: a) ectoparasites, that live on the body surface of the host; b) endoparasites, that live in their hosts’ organs, tissues, or cells and often rely on a third organism (the carrier, or vector) to transmit them to the host. The cuckoo and cowbird practise *brood parasitism*, laying eggs in other birds’ nests to be raised by the foster parents. In *social parasitism*, one type of animal parasitises animals of the same type (e.g., one ant species can parasite on other ant species). Hyperparasitism occurs when parasites are parasitised themselves (e.g., protozoans hyperparasitise fleas, that in turn, parasitise dogs).

In the EVM architecture, when a cell uses a computation from another cell, we call it symbiosis. Most spontaneously appearing symbiotic relationships are parasitic. Cells can easily parasite other cells, by using their computation. In the experiments we also observed instances of hyperparasites. It is common to parasite the parasites. These aspects will be discussed in detail in the experimental section, Chapter 9.

### 5.3.4 Cooperation

The layman view of evolution as “survival of the fittest” and “struggle for existence” in a gladiatorial perspective has been from the beginning of the Darwinian theory contrasted with the more settled collaborative and networking-views of natural selection, that promotes the cooperation and mutual aid. One of the strong opponents of strict competition-based interpretation of natural selection is Petr Kropotkin.

Happily enough, competition is not the rule either in the animal world or in mankind. It is limited among animals to exceptional periods, and natural selection finds better fields for its activity. Better conditions are created by the elimination of competition by means of mutual and mutual support. [...] One of the most frequent modes in which Natural Selection acts is, by adapting some individuals of a species to a somewhat different mode of life, whereby they are able to seize inappropriate places in Nature (Origin of Species, p145) – in other words, to avoid competition. [...] Don’t compete! – competition is always injurious to the species, and you have plenty of resources to avoid it! That is the tendency of nature, not always realized in full, but always present.

(Kropotkin, 1939, p.72-73)

Kropotkin’s conception of competition and “struggle for life as the struggle “for the greatest possible fullness and intensity of life with the least waste of energy” is different to the traditional conception of a brutally ruthless process that most Darwinians conceive it to be. The notions related to cooperation and relationship between competition and cooperation are discussed by contemporary biologists.
Competition makes for diversity, because, if I may speak figuratively, life is continually endeavoring to escape from it. In organized society the avenues of escape normally lead to activities requiring mutual aid and co-operation. Societies are mutual benefit associations and they tend to engender an altruism among their members that is quite closely subordinated to the egoistic interests of the group as a biological unit. One great advantage of social life is that it secures the benefits of competition and co-operation at the same time.

(Holmes, 1948)

‘The same holds true for the organism. In the organism there is a balanced interrelation between competitive and co-operative activities. Competition in this sense means that every component of the organism “strives” to achieve the same end, namely, the satisfactory functioning of the parts mutually involved. In this way the survival and adequate functioning of the organism as a whole is achieved.

(Montagu, 1952)

Natural selection taken exclusively as the increase in abstractly defined fitness is not adequate for all circumstances.

The unlucky substitution of ‘survival of the fittest’ for ‘natural selection’ has done much harm in consequence of the ambiguity of ‘fittest’ – which many take to mean ‘best’ or ‘highest’ – whereas natural selection may work towards degradation vide epizoa;14.

(Huxley, 1901).

The underlying theme in the above presentation is the present interdependence of organisms. Many biological models superimpose competition as an artificial concept over the highly complex web of relationships and mutual dependence (Bates, 1950).

5.3.5 Symbiogenesis

Proponents of symbiogenesis argue that symbiosis is a primary source of biological variation, and that acquisition and accumulation of random mutations alone are not sufficient to develop high levels of complexity (Margulis, 1970, 1981). Mereschkowsky (1905) and Wallin (1927) were the first to

14By epizoa Huxley meant the category of animals that live as parasites upon the exterior of the bodies of other animals. Such parasitic animals are frequently characterised by an extreme loss of structural complexity. In the earlier stages of their individual development they may be quite complexly organised creatures; in their adult stages their structural organisation may be reduced to the simplest form. Sacculina, the crustacean that belongs to the group of barnacles and attaches itself to the body of crabs, is a good example of such a parasite (Montagu, 1952).
propose that independent organisms merge (spontaneously) to form composites (new cell organelles, organs, species, etc). For example, important organelles, such as plastids or mitochondria, are thought to have evolved from an endosymbiosis between a Gram-negative bacterium and a pre-eukaryotic cell (Gray, Burger, and Lang, 2001; et al., 2003). A similar hypothesis can also be made regarding the origin of the nucleus. Global phylogenies of numerous protein sequences suggest that the ancestral eukaryotic cell arose by a unique endosymbiotic event involving engulfment of an eocyte archaeobacterium by a Gram-negative eubacterial host (Gupta and Golding, 1996). This is currently the most plausible explanation of emergence of a (complex) eukaryotic cell, and it diverges from typical neo-Darwinian views on the emergence of high complexity organisms.

Figure 5.3: Mereschkowsky’s drawing that illustrates how some bacteria lineages combined together through symbiosis and evolved together into more complex structures (Mereschkowsky, 1905).

In our EVM architecture, due to multiple concurrently operating independent evolutionary pres-
sures, it is possible to combine diverse self-contained cells (or programs) into new ones. This is one of the novel aspects and the significant contributions of our model, as compared with other traditional EC models. In traditional GA-like or Tierra-like systems, individual programs compete for resources and are not allowed to form symbiotic relationships. In contrast, in the EVM system, cooperation between cells or programs is the main driving force of the evolutionary progress. The majority of new evolutionary events are driven by a spontaneous formation of symbiotic relationships between individually evolving programs. The symbiogenetic aspects of our architecture will be discussed in more detail in later sections of this work.

5.3.6 Epigenetics

One of the important areas of evolutionary biology that extended substantially our understanding of the basic mechanisms of evolution is epigenetics (Waddington, 1942; Lederberg, 2001).

From the Greek prefix “in addition to”, epigenetics is the study of regulatory mechanisms that are not directly expressed in DNA sequences transmitted from one generation of cells or organisms to the next. Epigenetics is concerned with heritable changes in gene function that occur without a change in the sequence of DNA (Waterland and Jirtle, 2003). The main focus is here on the study of the processes involved in the unfolding development of an organism.

There is an observable variation among organisms that share identical DNA sequences, and historically it has been exclusively attributed to the effects of external environment. Contemporary studies confirm that environmental factors can have a strong effect on phenotype. The evidence from both animal and human experiments suggests that the impact of the environment has been overstated and that our views on the causes of phenotypic differences in genetically identical organisms require substantial revision (Wong, Gottesman, and Petronis, 2005).

In contemporary studies epigenetic mechanisms are often used to explain some puzzling phenomena in identical DNA animal studies. Epigenetics is employed to explain phenomena in cases where there are substantial phenotypic differences with almost no environmental changes between animals, and in cases where substantial environmental variations lead to almost negligible phenotypic changes.

Various aspects of the modern understanding of epigenetic inheritance are reminiscent of Jean-Baptiste Lamarck’s ideas about evolution. The theory of evolution proposed by Lamarck (established before the Darwinian formulation) has been discredited by the modern synthesis and gene-centric evolutionary models. However, in recent years there has been a development and some progress in understanding some of the epigenetic mechanisms that bring the original formulations of Lamarck back into the evolutionary biology agenda (Jablonka, Lachmann, and Lamb, 1992; Jablonka and Lamb, 2002). Simply stated, some of Lamarck’s original formulations are back into consideration and investigation.

The epigenome is the overall epigenetic state of a cell. As one embryo can generate a multitude of cell developmental trajectories, one genome could be said to give rise to many epigenomes. This
is somewhat similar to the notion of Rnomics, discussed in detail in the following section. Epigenetics refers to the phenotypic level. Most contemporary biologists acknowledge the epigenetic effects, however the exact extent and influence on evolutionary process is still unknown. The notion of the epigenome is making its way into mainstream genetics. The new emerging notions of the epigenome expose complex network-like structural and functional dependencies between different genetic structures. It seems that the genetic dependencies and information encoding and transfer may be much more complex than originally envisioned.

A simple gene-centred model (discussed in Section 5.2.3), that postulates an almost isomorphic relationship between genome and phenotype is being aligned with the new extended model that encompasses epigenetic effects.

The epigenome is not a structure that is likely to replace the role attributed by orthodox gene-centred evolutionary models (Murrell, Rakyan, and Beck, 2005) to the genome, but these should be used together to formulate more accurate and universal models of biological evolution. For sure, the epigenetic effects should not be neglected, but investigated in detail to establish their scope and importance.

An interesting experiment led by Robert Pruitt from Purdue University confirmed the violation of the orthodox Mendelian model of inheritance. In the experiments a single plant species was able to bypass a generation, and pass genetic information directly from grandparents to grandchildren, i.e. in spite of the mutant genes present in both parents, the offspring could reproduce the healthy DNA structure of their grandparents (Lolle1, Victor, Young, and Pruitt, 2005). Pruitt suggests that RNA structures somehow can be used as a secondary coding mechanism for inherited information. The plant, according to Pruitt, can allow the offspring to “try” some new mutations, and in the case of harmful effects, the offspring can revert its genetic make-up back to the one of its grandparents. Experiments and results like that put a questionmark on naive gene-centric and Mendelian models of inheritance and evolutionary processes.

In our EVM model we try to depart from naive and simplistic EC models based on neo-Darwinian notions and mimic the complex network-like dependencies. It is, however, too early to speculate to what extent our architecture can express and capture the biological dependencies.

5.3.7 Rnomics

There is growing scientific evidence (Vogel, Bartels, Tang, Churakov, Slagter-Jäger, Hüttenhofer, and Wagner, 2003; Hare and Palumbi, 2003) that introns\textsuperscript{15} i.e. non-coding RNA regions (colloquially referred to as non-coding DNA or junk DNA) are important components of a complicated genetic

\textsuperscript{15}Note, that the meaning of the term intron differs in contemporary genetics and in evolutionary computation (EC). In EC this term relates to a piece of code that is not executed, and therefore does not contribute anything to the function of the genome. In the context of recent studies and progress in evolutionary biology and genetics, the semantic meaning of the term as adapted in EC is somewhat misleading. It is speculated that introns in the biological sense play an important role in the overall behaviour and expression of the genotype.
network (Mattick, 2001). The evidence suggests that a cell uses numerous small non-protein-coding RNAs, including small nuclear RNAs (snoRNAs), microRNAs (miRNAs), short interfering RNAs (siRNAs) and small double-stranded RNAs. These regulate gene expression at many levels, including chromatin architecture, RNA editing, RNA stability, translation, and quite possibly transcription and splicing (Mattick and Makunin, 2005).

The experimental data from this area of genetics suggests that these non-coding RNA sequences, the DNA, mRNA and the proteins, all interact with each other, and that the introns act as nodes: link points in this extended genetic network. Introns provide many extra connections and link different parts of the network. It is worth noting that information theory (Shannon and Weaver, 1949) predicts that as the complexity of a cell’s protein machinery increases, the complexity of the systems required to coordinate its assembly and operation rises exponentially until it greatly exceeds that of the machinery itself (Mattick, 2005). There is growing evidence that RNA molecules have ultimate control over when and where genes switch on and off. By switching integrated networks of genes on, and switching others off, RNA sequences ultimately control cell identity. This also enables a form of multitasking and parallel processing within the genetic network (Mattick and Gagen, 2001).

Mattick says:

“It’s an amazingly robust and predictable process, self-organising and exquisitely accurate. People simplistically assumed there is sufficient information just in the combination of proteins and regulatory factors to roll out the embryonic program. It’s like expecting all the transistors, screws, rivets and other bits of a Boeing 747 to just assemble themselves.”

On the other hand, simple single-celled organisms, such as bacteria (prokaryotes) are characterised as having a severely limited number of introns. A bacterial gene simply codes for only one protein. Bacteria (prokaryotes) due to their low complexity level do not need such a complicated coordination mechanisms as observed in eukaryotic cells and multi-cellular organisms. Bacteria seem, then, to be coded more directly, and they are more rigid and inflexible. This is where genetic algorithms and evolutionary computation take their inspiration. This is also the source of the limitation of these systems. Our EVM architecture tries to overcome this limitation by providing mechanisms to coordinate and join interactions of different code components within and across different levels.

Multi-cellular organisms, as pointed out above, utilise the network-based parallel architecture of coding and non-coding regions of DNA (Cooper, 2002; Mattick and Gagen, 2001). The intron-divided modules of genes in higher organisms can be mixed and recombined during RNA editing to create multiple proteins with different functions. The same editing process can produce multiple RNA structures. According to the Rnome model, the Rnome is the major source of human individuality. It is not just the genes, but mostly the functional control structures in the form of non-coding RNA sequences. Non-coding DNA offers special mechanisms for mutation relative to the tiny amount of DNA devoted to genes. Disturbances in the RNA networks, affecting how genes respond to envi-
From (Mattick and Gagen, 2001):

We suggest that these RNAs have evolved to function as endogenous network control molecules which enable direct gene-gene communication and multitasking of eukaryotic genomes. Analysis of a range of complex genetic phenomena in which RNA is involved or implicated, including co-suppression, transgene silencing, RNA interference, imprinting, methylation, and transvection, suggests that a higher-order regulatory system based on RNA signals operates in the higher eukaryotes and involves chromatin remodeling as well as other RNA-DNA, RNA-RNA, and RNA-protein interactions. The evolution of densely connected gene networks would be expected to result in a relatively stable core proteome due to the multiple reuse of components, implying that cellular differentiation and phenotypic variation in the higher eukaryotes results primarily from variation in the control architecture. Thus, network integration and multitasking using trans-acting RNA molecules produced in parallel with protein-coding sequences may underpin both the evolution of developmentally sophisticated multicellular organisms and the rapid expansion of phenotypic complexity into uncontested environments such as those initiated in the Cambrian radiation and those seen after major extinction events.

Our EVM architecture is closely related to this highly parallel inter-connected and cross-linked model of genetics. Although originally designed without knowledge of or any relation to the most recent advances in genetics and Rnomics in particular, the EVM architecture predicts certain features that are now being confirmed by contemporary genetics. The EVM opens doors for further research in evolutionary biology, and promises advance in the field of evolutionary computation, by breaking the limits of simple single-cellular organisms, such as bacteria. The EVM model shares its architecture with the more complex Rnomic model of contemporary genetics. The difficulties observed in evolving EVM programs may provide some insights into the time frames of biological evolution, and in particular, explain why it took so long for the first eukaryotic cells to appear on Earth.

5.3.8 Specialisation

Another phenomenon widely spread in nature, that occurs at all levels of biological organisation from molecules to populations, is specialisation. Specialisation is the process of setting apart a particular sub-system (reducing its complexity) for better efficiency of a particular function. As an example, the cells of a vertebrate body exhibit more than 200 different modes of specialisation (Alberts, Bray.

---

16The original EVM architectural model dates back to 1998, and authors were unaware of the progress in Rnomics and sub-genetic control mechanisms till late 2005.
A non-exhaustive list of these, as well as those occurring at lower levels, is shown in Table 5.1. Our working hypothesis is, that specialisation together with symbiosis is necessary to reach higher complexity levels.

Table 5.1: Some examples of specialisation at molecular, organelles, and cellular levels. DNA: deoxyribonucleic acid; RNA: ribonucleic acid.

<table>
<thead>
<tr>
<th>Molecular level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA, RNA</td>
<td>Information carrier</td>
</tr>
<tr>
<td>RNA, protein</td>
<td>Catalytic function</td>
</tr>
<tr>
<td>protein, phospholipids</td>
<td>Structural components</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Organelles level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>Genetic information storage</td>
</tr>
<tr>
<td>Rough endoplasmic reticulum</td>
<td>Secreted or membranous protein synthesis</td>
</tr>
<tr>
<td>Smooth endoplasmic reticulum</td>
<td>Lipid metabolism, detoxification</td>
</tr>
<tr>
<td>Golgi apparatus</td>
<td>Macromolecule processing</td>
</tr>
<tr>
<td>Endosome</td>
<td>Vesicle trafficking</td>
</tr>
<tr>
<td>Lysosome</td>
<td>Intracellular digestion</td>
</tr>
<tr>
<td>Mitochondria</td>
<td>Energy production (ATP)</td>
</tr>
<tr>
<td>Chloroplast (plants only)</td>
<td>Photosynthesis</td>
</tr>
<tr>
<td>Peroxisome</td>
<td>Oxidative reaction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cellular level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Neuronal cell</td>
<td>Signal transmission</td>
</tr>
<tr>
<td>Muscle cell</td>
<td>Contraction</td>
</tr>
<tr>
<td>B and T lymphocyte, natural killer cells, denditic cells</td>
<td>Antibody secretion, cytolytic effect, antigen capture</td>
</tr>
</tbody>
</table>

Figure 5.4 shows some specialisation examples at the cellular level.

According to the orthodox gene-centred view of evolutionary processes, children from a given culture should develop a specialisation towards a specific language, simply by the process of natural selection. In Japan, for example, due to a long separation from other influences, it is phenotypically useless to have broader capabilities, that are going to be disposed of in the first months of the baby’s life. However, this is simply not the case, and a simple neo-Darwinian models fail to capture the fact that children around the world continue to maintain the universal phonetic language capabilities when born. These capabilities are specialised during the first few months of infants’ development. Let us consider the ability to understand and produce speech (Lenneberg, 1967). It has been noted
that infants (up to about 3-6 months old) can discriminate and produce sounds from many different languages in the world, regardless of their sex and ethnicity. These capabilities are lost during the infant development process. Infants adapt to the social group they are being brought up in, and they specialise only to those sounds that are communicated to them. The capability to discriminate and produce sounds of any language are lost. For example young Japanese infants can easily distinguish “r” and “l” sounds, in words such as “fry” and “fly”. This ability is almost completely lost in Japanese adults, both, to hear and discriminate between these two, and to produce appropriate sounds. It is claimed that this loss serves an important function since our information-handling capacity is limited. It is argued that it is not so much the characteristics of the sounds, but the number of distinct states that need to be assimilated and learnt. The fewer distinct sounds, the easier the baby will acquire the language.

It is our belief that specialisation is a central element of biological evolution that has been overlooked in simple artificial EC models. The presence of specialised molecular subsets was absolutely necessary to the formation of the simplest living organism. What is really interesting, however, is how higher complexity levels have been reached in the course of evolution.

In our EVM models we always start with broader capabilities that are trimmed to fit a specific purpose. It is not only that the evolutionary processes specialise species – it is species that are evolved to specialise themselves too. In the context of EVM programs, the more specialised the virtual machine is, the easier it is to acquire the necessary performance and efficiency of a given program. The smaller the instruction set, the easier is to learn or to find a suitably optimised program.
5.4 Stem cells

In the EVM architecture we use the notion of a cell and a collection of cells. We sometimes use the intuitions of cell development. Some of the notions are based on stem cells. Initially, all the EVM cells are not specialised, just as stem cells are not specialised. One of the fundamental properties of a stem cell is that it does not have any tissue-specific structures that allow it to perform specialised functions. A stem cell cannot work with its neighbours to pump blood through the body (like a heart muscle cell); it cannot carry molecules of oxygen through the bloodstream (like a red blood cell); and it cannot fire electrochemical signals to other cells that allow the body to move or speak (like a nerve cell). However, non-specialised stem cells can give rise to specialised cells, including heart muscle cells, blood cells, or nerve cells.

Stem cells are capable of dividing and renewing themselves for long periods. Unlike muscle cells, blood cells, or nerve cells (which do not normally replicate themselves) stem cells may replicate many times. When cells replicate themselves many times over, it is called proliferation. If the resulting cells continue to be unspecialised, like the parent stem cells, the cells are said to be capable of long-term self-renewal.

Stem cells can give rise to specialised cells. When unspecialised stem cells give rise to specialised cells, the process is called differentiation. We use the term differentiation and cell specialisation interchangeably in the context of the EVM architecture. In biological cells the internal signals are controlled by a cell’s genes. The external signals for cell differentiation include chemicals secreted by other cells, physical contact with neighbouring cells, and certain molecules in the micro-environment.

It is not completely understood why stem cells remain unspecialised in the organism, and what are the signals in a mature organism that cause a stem cell population to proliferate and remain unspecialised until the cells are needed for repair of a specific tissue. It is also not understood how the differentiation mechanisms actually work in biological cells. It is an interesting area of research, and the EVM architecture provides a framework for different differentiation models. Although, it was not the main focus of the current study, we have investigated certain models of cell specialisation, and provided experimental analysis between different models.

5.5 The Baldwin effect

One related issue to genetic information transfer is research on the Baldwin effect. Baldwinian evolution, also called ontogenic evolution, is a theory proposed by James Mark Baldwin that states that organisms can pass on learnt abilities to their offspring, through a selection for learning abilities. Selection pressure promotes offspring that are more phenotypically plastic, offspring that have increased capacity for learning new skills or new functions, and effectively would learn skills previously learnt by their ancestors faster. Although originally formulated in connection with neo-Darwinism, the relation of Baldwinian evolution to epigenetics and Rnomics is not well investigated and largely unknown.
at this point. It is possible that further research in Rnomics and epigenetic mechanisms of inheritance will draw these separate theories closer together.

In the context of the EVM architecture and multi-task learning, it is clear that elastic and flexible solutions capable of coping with multiple tasks are more successful and are promoted, even though there may not be any higher level pressure for that provided from outside. It is just the interactions of all the low-level rewards that push the evolution of EVM programs towards more and more capable solutions; because they simply benefit multiple niches and resources, yielding better overall benefits to solutions specialising in single, fixed tasks. Our experiments suggest also that highly variable environments spontaneously promote more flexible and adaptable programs, and fixed and steady environments promote simpler and more rigid designs.

The ability to learn, and to be adaptable, co-operative and have the ability to specialise, are all general traits that are normally not present in evolutionary computing and in orthodox gene-centred views. These are the ingredients that are missing in many contemporary evolutionary computation models. Orthodox gene-centered model do not use adaptive, co-operative and niching techniques all at the same time, even though there is a number of active research in each of these areas individually. They are ingredients that the EVM architecture tries to unify, exploit and benefit from. Some evolutionary biologists try to argue that those general traits are beneficial, therefore they have evolved using traditional evolutionary models. This however has not been replicated in laboratory settings. Gene-centred EC models do not evolve those traits, even if they are beneficial, simply because evolving specialised solutions is cheaper and faster. Specialised solutions usually outperform the generic, more flexible designs.

It is not known why and how these general traits evolved. The EVM model is a step towards better understanding of these and related phenomena. Due to the nature of the EVM search, multiple tasks can be accommodated by a single computational cell. Similar to the work of Schmidhuber (Schmidhuber, 1999), we employ multi-task learning techniques that promote compact representations capable of solving multiple related tasks. The precise architectural and operational differences and similarities between the EVM model and contemporary EC systems are in Chapter 6.

5.6 Neutral evolution

5.6.1 Genetic drift

In a simplified view, an individual in the neo-Darwinian theory of evolution is represented as a sequence of alleles. Genetic drift is a stochastic process that arises from the role of random sampling in the production of offspring of such individuals. The genes of each new generation are not a simple copy of the genes of the successful members of the previous one, but rather a sampling, which includes some statistical error. Drift is the cumulative effect over time of this sampling error on the allele frequencies in the population. Genetic drift has no preferred direction. Any neutral allele may
be expected to increase or decrease in any given generation with equal probability. Over a long time each of the neutral alleles will either die out or be present in 100% of the population, there will be no random variation of the given gene, and all the instances will become uniform. Thus genetic drift tends to sweep gene variants out of a population.

A detailed explanation of drift is given in the following excerpt from Wikipedia:

From the perspective of population genetics, drift is a sampling effect. To illustrate: on average, coins turn up heads or tails with equal probability. Yet just a few tosses in a row are unlikely to produce heads and tails in equal number. The numbers are no more likely to be exactly equal for a large number of tosses in a row, but the discrepancy in number can be insignificantly small (in statistical terms). As an example, ten tosses turn up at least 70% heads about once in every six tries, but the chance of a hundred tosses in a row producing at least 70% heads is only about one in 25,000.

Similarly, in a breeding population, if an allele has a frequency of \( p \), probability theory dictates that (if natural selection is not acting) in the following generation, a fraction \( p \) of the population will inherit that particular allele. However, as with the coin toss above, allele frequencies in real populations are not probability distributions; rather, they are a random sample, and are thus subject to the same statistical fluctuations (sampling error).

When the alleles of a gene do not differ with regard to fitness, on average the number of carriers in one generation is proportional to the number of carriers in the previous generation. But the average is never tallied, and therefore the frequency of an allele among the offspring often differs from its frequency in the parent generation. In the offspring generation, the allele might therefore have a frequency \( p' \), slightly different from \( p \). In this situation, the allele frequencies are said to have drifted. Note that the frequency of the allele in subsequent generations will now be determined by the new frequency \( p' \).

As in the coin toss example above, the size of the breeding population (the effective population size) governs the strength of the drift effect. When the effective population size is small, genetic drift will be stronger.

More information about the mechanism of genetic drift is provided in Coyne and Orr (2004).

### 5.6.2 The neutral theory of molecular evolution

The neutral theory of molecular evolution is an evolutionary theory formulated and proposed first by Kimura (1968, 1983). The theory states that the main driving element of evolution is the genetic drift. It is sometimes claimed that the role of genetic drift is far stronger than that of natural selection (King and Jukes, 1969).
According to the theory, the vast majority of molecular differences are selectively neutral. These differences that are considered neutral by definition do not influence the fitness of either the species or the individuals who make up the species. Therefore, the theory regards these genome features as neither subject to, nor explicable by, natural selection. The theory states that through drift, new neutral alleles may become more common within the population. They may subsequently decline and disappear, or in rare cases they may become fixed\(^1\) – meaning that the substitution they carry becomes a universal feature of the population or species. When an allele carrying one of these new substitutions becomes fixed, the effect is to add a substitution to the sequence of the previously fixed allele. In this way, neutral substitutions tend to accumulate, and genomes tend to evolve.

### 5.6.3 Neutral computational evolution

Neutral theory of molecular evolution is relevant to the field of evolutionary computation, primarily due to the fact of sampling effects of finite size discrete populations.

The theory of neutral evolution is a subject of experimental and theoretical study of certain models within evolutionary computation. The effects of genetic drift are known to the EC community, and we will discuss some of the models that are inspired by the theory in Section 6.5.

### 5.7 The EVM biological inspirations

In this chapter we have presented various biological theories that extend neo-Darwinism and provide a more accurate account for biological evolutionary models. Evolution from a biological perspective is a complex and not yet fully understood phenomenon. The EVM model draws from and extends the basic Darwinian notions and concentrates on three main extensions: specialisation, symbiosis, and exaptation. There are other concepts employed within the EVM architecture, which have direct or indirect linkages with contemporary theories within the fields of epigenetic, Rnomics, and neutral evolutionary theories.

In Chapter 6, we discuss the contemporary computational evolutionary models. We concentrate on their biological inspirations and implications and provide a discussion of their limitations. Then, in the following Chapter 7, we will discuss in detail the notions of autopoiesis and hypercycles. This will complete the broad review of the current state of the art in the areas of biological and computational evolutionary research. The following chapters will describe the EVM experiments and provide the experimental results.

\(^{1}\)In the EVM architecture we use the term to freeze a given program sequence. We use that term however both for selectively advantageous and neutral sequences of code.
Chapter 6

Artificial Evolution
The human mind delights in finding pattern – so much so that we often mistake coincidence or forced analogy for profound meaning. No other habit of thought lies so deeply within the soul of a small creature trying to make sense of a complex world not constructed for it.

(Gould, 1987b)

6.1 Introduction

In Chapter 4, we have discussed different models of computation and we have investigated possible models suitable for modelling naturally occurring phenomena. In this chapter, we are interested in the exact process of replicating complex biological systems in computational realms. In Chapter 5 we have discussed different biological aspects of evolutionary theory. We have introduced several different concepts and intuitions from the biological realms. We are now going to explore them further in this chapter in the context of computational evolutionary systems. We will show how we have adapted some of the biological concepts to augment the current state-of-the-art in evolutionary computation when designing and developing our EVMA. We focus on the notions of symbiogenesis, exaptation, rnomics and regulatory mechanisms of genetic networks.

In this chapter, we present the current state-of-the-art in Evolutionary Computation (EC) and the field of Artificial Life (ALife). In Chapter 7 we will present efforts that tackle the same problems, but efforts that have been initiated in other disciplines. These are based on theory of hypercycles and autopoiesis. Those various models together augment and complement each other. They largely deal with the same underlying phenomena of self-assembling and self-organising homeostatic ensembles. We focus on possible models for life-long reinforcement learning in chapter 8. All the discussion will be based on the EVMA presented in Chapter 2, together with a concrete EVMI described in Chapter 3.

This chapter is organised as follows: first we discuss general models of artificial evolution and ALife. Then, we concentrate on the analysis of different computational models of evolutionary processes and homeostatic ensembles that we found interesting and relevant to the concepts discussed previously in Chapter 5. In other words, we focus on those computational models that are most relevant to the biological models of evolution as discussed in the previous chapter. Those are models, that in some way mimic the properties and follow closely the biological theory of evolution. Then we present some of the advanced features of evolutionary computation methods, followed by a discussion of their strengths and limitations.
6.2 Artificial evolution and artificial life

Artificial evolution is the umbrella term denoting different models of computational evolution, models of evolution inspired by biological evolution. Artificial life (ALife) is the study of life through the use of computer models, synthesis and simulations. Computer scientist Christopher Langton coined the term Artificial life when he held the first “International Conference on the Synthesis and Simulation of Living Systems” (Artificial Life I) at the Los Alamos National Laboratory in 1987:

Artificial life is the study of artificial systems that exhibit behavior characteristic of natural living systems. It is the quest to explain life in any of its possible manifestations, without restriction to the particular examples that have evolved on earth. This includes biological and chemical experiments, computer simulations, and purely theoretical endeavors. Processes occurring on molecular, social, and evolutionary scales are subject to investigation. The ultimate goal is to extract the logical form of living systems.

The EVMA model itself can be seen as belonging to the ALife category, however, our main intention is to build a model of artificial evolution and we use the term evolutionary computing with respect to the EVM architecture, too. In the following sections, we will review the basic models related to both artificial life and evolutionary computing models. We start with simple automata models for self-replication and self-organisation.

6.3 Automata models

In some ways, artificial life and artificial evolution, under different labels, became a subject of investigations shortly after the first formal models of computation were proposed (Turing, 1948). Towards the end of his life, Turing himself concentrated on mathematical biology, in particular, morphogenesis. He investigated the existence of Fibonacci numbers in plant structures. He used reaction-diffusion equations which are now central to ALife in the area of pattern formation. Turing’s contributions span multiple interest areas. They nicely demonstrate that computational biology started almost simultaneously with the inception of a theory of computing. Turing’s work is an excellent example of how the computing theory can be used in modelling and understanding naturally occurring phenomena in biology.

Another important incarnation of early ALife systems is the area of cellular automata. A cellular automaton is a computational model composed of a number of simple and spatially organised computational interconnected cells. The cell’s state can be communicated to its neighbours. The state of a given cell is a function of the cell’s state and the state of its neighbours. John\textsuperscript{1} von Neumann (von Neumann, 1963; von Neumann and Burks, 1966), Stanisław Marcin Ulam, and later John Conway are considered the main influential figures shaping the cellular automata landscape in the 1950s and

\textsuperscript{1}Johannes Luis von Neumann is widely known as John von Neumann
Figure 6.1: An example view of Conway’s Game of Life cellular automata.
1960s. Stanisław Ulam and von Neumann used computers at Los Alamos to produce a handful of examples of cellular automata. Ulam investigated so called recursively defined geometric objects. These are essentially the results of evolving generalised 2-D cellular automata from a single block of black cells (Wolfram, 2002). Ulam noted that in some cases simple growth rules are capable of generating a complicated pattern. Despite the intuition that these results may be relevant to biology, no progress was made on this with traditional mathematical methods and the results were not widely known. The work of von Neumann and Ulam bootstrapped the entire cellular automata field. Contemporary researchers, for the most part, abandoned the mathematical and analytical research methods, and focus on using computational models. The properties and various phenomena associated with CA are explored through direct experimentation and computational modelling (Wolfram, 2002).

6.3.1 Self-replication

John von Neumann (von Neumann, 1963; von Neumann and Burks, 1966) was most likely the first person trying to develop an abstract model of biological self-reproduction (a topic that emerged from investigations in cybernetics). Around 1947 von Neumann began thinking about models of self-reproducing robots, with access to loosely organised part repositories. After suggestions from Stanisław Ulam (in 1951), von Neumann turned into more abstract mathematical models, and ended up with a 2-D cellular automaton model of self-replication. His particular cellular automaton constructed in 1952-53 had 29 possible colours (states) for each cell. The state transitions were governed by a complicated set of rules. The rules were inspired by and emulated the operations of components of various electronic and mechanical devices. To give a mathematical proof of the possibility of self-reproduction, von Neumann then outlined the construction of an impressive 200,000 cell configuration, which was capable of reproducing itself.

Von Neumann appears to have believed – presumably in part from seeing the complexity of actual biological organisms and electronic computers – that something like this level of complexity would inevitably be necessary for a system to exhibit sophisticated capabilities such as self-reproduction.

(Wolfram, 2002), page 876.

Further research in this area demonstrated that the actual complexity of a self-reproducing automata model does not require such an overwhelming complexity level. The work of Michael J. Apter’s work (Apter, 1964, 1969) is an example of the earliest attempts in self-replicating automata, exhibiting universal computation. Apter closely followed the model described by von Neumann (von Neumann, 1963; von Neumann and Burks, 1966), but proposed a much simpler scheme for self-reproduction. His main motivation was to propose a model of replicating Turing machines (or any finite state automata) of an arbitrary shape and size. Apart from demonstrating growth of a structural relationships, the model has not been used to solve any problems or as a learning algorithm. Apter’s work is the
one of the earliest attempts to capture the properties of growth and self-replication in the Universal Turing machine environment. Note, that our EVM architecture also fits into this category, trying not to sacrifice the Universal Turing machine computation.

There are other models of self-replication, for example replicating structures of certain cellular models. The famous Langdon’s loops are a good example (Langton, 1984, 1986). Langton’s automaton uses eight states and twenty-nine rules. The structure that reproduces itself is a loop-structured pattern on a 2-D grid. Similar to von Neumann’s automata, Langton’s loops demonstrate one of the fundamental properties of living organisms: self-reproduction. The important observation is that complex behaviour such as self-reproduction can be explained in terms of interactions of simple elements and that it can be studied in its logical formulation independently of its physical realisation.

### 6.3.2 Growth models

Michael Apter (Apter, 1964, 1969) was most likely the first person trying to use computers for biological simulations. He adapted the Turing machine model of automata for the description of growth systems of various kinds. He used the von Neumann model of reproduction, and he investigated models where the amount of growth is determined by the number of parts available – modelled through the availability of computational resources.

The Apter’s model consists of the usual set of constructs necessary for a working Turing machine, plus one extra element: the standard alphabet of symbols (for read/write operations) has been extended by a special possible output $R$. $R$ represents the copy of the current automaton. The copy possess an identical instruction set and state set. The copy is inserted directly behind the current automaton. Each automata tape is assumed to move continually in one direction between its respective machines at the rate of one square of the tape per time unit. It is also assumed that it takes only one moment for a square of tape to pass between automata. Each state transition of an automata moves its head one tape unit to the right. Each new automata’s initial symbol read from the tape is $S_0$.

The following (Figure 6.1) is the example of a growth model, where an initial automaton produces a copy of itself. We use a simple notation of the form $q_i S_i S_j q_j$, i.e. when in state $q_i$ and scanning symbol $S_i$ from the tape, write symbol $S_j$ onto the tape and change to state $q_j$.

The evolution of the growth model, from the single automata, with the initial symbol on the tape $S_0$ and initial state $q_1$ is presented in Figure 6.2.
Figure 6.2: Growth models. Apter model for growing a line of 2 automata.

Table 6.2: Apter’s growth model, for line of 3 automata.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$S_0$</td>
<td>$R$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$S_0$</td>
<td>$S_0$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

161
Table 6.3: Apter’s growth model, for line of 5 automata.

\[
\begin{align*}
q_1 & S_0 \rightarrow R q_2 \\
q_1 & S_1 S_0 \rightarrow q_6 \\
q_2 & S_0 \rightarrow S_0 q_3 \\
q_2 & S_1 S_1 \rightarrow q_6 \\
q_3 & S_0 \rightarrow S_0 q_4 \\
q_3 & S_1 S_1 \rightarrow q_6 \\
q_4 & S_0 q_0 \rightarrow q_5 \\
q_4 & S_1 S_1 \rightarrow q_6 \\
q_5 & S_0 S_1 \rightarrow q_6
\end{align*}
\]

Table 6.4: Apter’s growth model, X-shaped 5 automata.

If one imagines the connectivity of the automata in 3-D space, where a single tape can be split into two (or more) independent tapes, Apter’s growth model can be used to build other spatially structured automata. Consider for example the X-shaped 5 automata model, Table 6.4; and Y-shaped 5 automata model, Table 6.5.

6.4 Evolutionary computation

*Evolutionary computation* (EC) is generally considered a sub-field of artificial intelligence, parallel but with some overlap with Artificial Life. EC is a loosely defined term that denotes any biologically inspired computational methods applied to optimisation. It may be based on biological theory of evolution and/or genetics, but it is not limited to any particular biological theory. EC often involves meta-heuristic optimisation algorithms, such as:

- Genetic algorithms (Goldberg, 1989), genetic programming (Koza, 1992). These will be discussed later in more detail.
Evolution strategy (ES), is an optimisation technique that uses real-vector coding, mutation, recombination, and selection as its primary operators. Sometimes ES is limited purely to special mutation, and the variance of the distribution is adapted during the optimisation (Schwefel, 1993; Beyer and Schwefel, 2002).

Evolutionary programming (EP), is an optimisation technique that is similar to genetic programming, however, in EP the structure of the program is fixed and only its numerical parameters are allowed to evolve (Fogel, 1998).

Others, e.g. ant colony optimisation and swarm systems, particle swarm optimisation, and others (Bonabeau, Dorigo, and Theraulaz, 2000).

The Evolutionary algorithm is a generic term used to indicate any evolutionary computation model that is characterised by one, or more of the following properties: population-base, biologically inspired reproduction, genetic mutation, genetic recombination, and natural selection (based on the principle of survival of the fittest). Such a taxonomy is not strict, and many contemporary methods within the field of evolutionary computation overlap. It is difficult to clearly classify a given method exclusively to a single category. The reader should treat this taxonomy and terminology only as a guide, not a strict and rigid classification.

To present the main differences between the mainstream of contemporary methods of evolutionary computation and the EVM model, we will discuss the appropriate aspects of the existing methods and contrast them with the EVM abstract architecture. With some minor exceptions (that will be discussed in more detail later in this section), almost all contemporary evolutionary computing systems are derived from orthodox neo-Darwinism, and are descendants of a simple isomorphic mapping between the genotype and the phenotype (Sipper, 1999). The EVM architecture departs from this orthodox view and provides a more flexible, but complex, way of mapping genotype and phenotype. In fact, in

Table 6.5: Apter’s growth model, Y-shaped 5 automata.

- $q_1 S_0 R q_2$
- $q_1 S_1 S_0 q_6$
- $q_2 S_0 S_1 q_3$
- $q_3 S_0 R q_3$
- $q_3 (S_0, S_0) S_1 q_4$
- $q_4 S_0 R q_4$
- $q_4 (S_0, S_0) S_1 q_5$
- $q_5 S_0 R q_5$
- $q_5 (S_0, S_0) S_1 q_6$
the EVM model there is no fixed clear boundary between these two concepts as such. The boundary is blurred, because different pieces of code on different levels and at different times can influence each other and influence the overall behaviour of the complex network-like execution of the code. On an abstract level, the EVM is a meta-evolutionary computational model. It is our intention that EVM is capable of expressing and capturing a wider range of various biological models of evolutionary processes.

The practical and experimental aspects of the architecture depart in some places from the abstract view. These issues have been partially addressed in the Chapters 2 and 3. Further discussion on these issues are spread across various chapters throughout this thesis.

In the following subsections we will discuss in more detail some of the relevant subfields of EC in the context of EVM research. We also present some of the existing precursors of the EVM model, and compare their properties to the abstract EVM architecture.

6.4.1 Genetic Algorithms

The seminal work of John Holland (Holland, 1975), and the development of the theory of genetic algorithms (GAs) was a founding event in evolutionary computation. The biological theory of evolution based on neo-Darwinism fascinated and inspired Holland:

> At the start, my interest was primarily mathematical, sparked by Fisher’s remarkable book "The genetical theory of natural selection". For me it was a revelation that you could use sophisticated mathematics to study biology. The shortcoming of Fisher’s theory, that you could only treat genes as additive, sparked my interest in co-adapted sets of genes (schemata) as a way of generalizing his fundamental theorem. From there it wasn’t much of a leap to realize that natural selection solved really complicated nonlinear problems. Because almost all attempts at solving nonlinear problems were via linear approximations, this alternative seemed worth exploring, despite remarks about the "slowness of evolution" (a complete misunderstanding of the process). As a final comment, it seems to me that many, in and out of biology, do not yet understand the power of recombining building blocks.

Genetic algorithms are typically implemented as a population of solution representations, called *individuals*. The evolution starts from a *population* of completely random individuals, and the individuals are subjected synchronously to series of *genetic operators*, inspired by biological genetics. These operators are: mutation, selection, mating, and crossover. Traditionally, solutions are represented as binary strings, but different representations are also being used, the most popular are integer coding and strings of alphanumeric characters. If the genetic operators are being applied asynchronously and influence only a subset of the population, this is usually referred as a *steady-state model* (Vavak and Fogarty, 1996). If the population is divided into demes (subpopulations), the processing is kept
local to individual demes, and the mixing between demes is, for example, obtained by migration operator, this is called a multi-deme genetic algorithm. There are many other possible models of parallel processing within the genetic algorithm paradigm (Nowostawski, 1998; Nowostawski and Poli, 1999).

Despite the initial fascination and hopes, the work within the field of GAs and variations of EC has not delivered the silver bullet\(^2\) for automated learning methods. The field blossomed with variety of different models and forms of GA used for more and more elaborate optimisation problems. The original theory has been modelled analytically with great detail (Vose, 1999) and then extended into different computational frameworks (Fogel, 1998).

All GA variants share the same principles inspired by the modern synthesis in evolutionary biology. All the computational methods use the gene-centric model as a basis. They use single encoding for the genotype-phenotype mapping, and they use externally defined single fitness function to exercise external selection pressure.

A given problem encoding a set of parameters, and the fitness function itself determine the efficiency of the search process. The best performing GAs have properly encoded individuals working in a smooth fitness landscape, with long slopes leading to wide and flat global optima (simple-hill climbing being the ultimate goal of the encoding). In extremely “rough and rugged terrain”, with extremely short slopes and sharp peaks, with highly discontinuous fitness landscapes, GAs perform poorly, or, cannot perform at all (Korkmaz, 2003). Such a fitness terrain is typical in computer programs, where a single point mutation can dramatically alter a fitness value of a long programme sequence. For such landscapes various techniques must be employed to help with the evolutionary search process. Evolutionary techniques for evolving computer program are discussed in more detail in the following Section 6.4.2.

### 6.4.2 Genetic Programming

One of the areas of evolutionary computation is a particular model of genetic algorithms aimed at evolving computer programs. This sub-field, dedicated purely to the task of evolution of executable computer programs, is referred to in the literature broadly as genetic programming. Genetic programming (GP) is an automated methodology based on genetic algorithms, designed to find executable computer programs performing a task specified by the user. Genetic programming was first investigated by Stephen F. Smith (Smith, 1980) and independently by Nichael Cramer in 1985 (Cramer, 1985). In his studies Cramer explored the use of GA to evolve both tree-structured and string-based

\(^2\)The metaphor of the silver bullet applies to any straightforward solution perceived to have extreme effectiveness. The phrase typically appears with an expectation that some new technology or practice will easily cure a major prevailing problem. Traditionally, the silver bullet is the only weapon that can kill a witch, vampire, monster, or a person living a charmed life. There is a well-known paper (Brooks, 1986) on software engineering, where Brooks argues that there will be no more technologies or practises that will serve as ”silver bullets”. Note also, this is related to the concept of No Free Lunch discussed earlier.
computer programs. Independently of Cramer work, Schmidhuber investigated similar algorithms on a PROLOG-based GP system directly operating on variable-length code (Dickmanns, Schmidhuber, and A.Winklhofer, 1987). Later, GP models based on LISP and tree-based representations were extensively studied by Koza (Koza, 1992). Koza used exclusively tree-based representations (and Lisp as a programming language), and his work is influential within the field. Koza popularised the tree-based GP and is considered a father of contemporary tree-based GP methods (Banzhaf, Nordin, Keller, and Francone, 1997).

Evolved programs can be expressed in many languages, with different syntax and semantics. As mentioned before, the most prevalent is the tree-based representation (Koza, 1992), based on simple arithmetical and logical set of operators, applied for different explorations of the algebraic and Boolean logic search spaces. The subset of GP where computer programs are represented by a linear structure are referred to as linear genetic programming. Contemporary GP is probably evenly divided between tree-based and linear representations. There has been work done in machine code representations (Nordin, 1994; Nordin, Banzhaf, and Francone, 1999), graph representations (Teller, 1996), linear-tree representation (Kantschik and Banzhaf, 2001a), linear-graph representation (Kantschik and Banzhaf, 2002), strongly-typed GP (Montana, 1995), stack-based GP (Perkis, 1994) and others.

There are many comparative studies between tree-based and linear GP program representations, e.g. (Kantschik and Banzhaf, 2001b). More information about the foundation of GP, theoretical analysis of the theory of building blocks and tree-based variants of GP can be found in Langdon and Poli (2001).

One of the important aspects of GP is the notion of abstraction and modular organisation of the generated code. Those features facilitate the search for solutions to complex problems. However, standard GP does not provide any mechanism for data abstraction and automatic code reuse. After the initial concepts of automatically defined functions (ADFs) (Koza, 1992), there have been many different attempts to extend GP and facilitate more robust and flexible methods of abstraction and code reuse (Rosca and Ballard, 1994).

Considerable research efforts have been dedicated to the efficient search for global optima (or set of local optima) in a predefined fitness landscape. Such efforts are useful steps towards a better understanding of the process of evolution and evolvability in general. However, there are many issues concerning evolvability that go beyond the task of finding a global (or local) optimum. There are several branching research areas that spin off the mainstream of genetic programming efforts, three of which will be discussed in detail below. Although our EVM architecture does not directly share much in common with the original GA and GP models, it takes its inspiration from the work of some researchers in the field of GP, in particular Roland Olsson, Conor Ryan and Michael O’Neill. We briefly introduce the three systems and point out the similarities to our EVM model. The EVM architecture shares some of the design and architectural decisions with the three systems discussed below.
6.4.3 Grammatical Evolution

One of the branches of research within GP algorithms is the use of context-free grammars to guide the process of the generation of solutions. The early efforts in this area used a grammar to represent the description of possible forms of a solution, or to represent legal problem states that can occur during the learning process e.g. LEX (Mitchell et al., 1984). One of the early examples is also the GREND-EL system, within the sub-field of inductive logic programming applications (Cohen, 1993), which used context-free grammars to represent the possible program structures. These models became an inspiration for the work of Peter Whigham (Whigham, 1995), who used context-free grammars together with the tree-based structured Genetic Programming model (Koza, 1992). Whigham advised a modification to the standard tree-based GP system, in which the context-free grammar was constraining the possible points of crossover and the behaviour of the mutation operators. This system tried to combine and constrain the normal GP search process through the use of some inductive heuristic process. This heuristic process effectively guides the selection and random variations within the search mechanisms of a normal tree-based GP.

These methods and extensions try to utilise smart handling and automated adapting of the search bias for a given problem. Several extensions have been proposed to facilitate the bias to be learnt during the search process. The bias is stored by an actual structure containing the production rules of the grammar, and by the numerical parameters attached to each of the alternative productions, specifying the merit of a given rule. The merit is then used in the decision-making process during the generation of the program to guide the choice towards the most promising program structures.

A direct descendant of the context-free grammar genetic programming system is a system called Grammatical Evolution (GE) developed by Conor Ryan, Michael O’Neill and J.J. Collins (Ryan et al., 1998; O’Neill and Ryan, 2003). This system can be used to automatically generate programs in an arbitrary language through simple finite state machine-based mechanisms and a language grammar specified in advance. Although GE provides much more flexibility in terms of coding and program representation, it is still based on the GP fixed-evolutionary model, and unlike more adaptive systems, it cannot self-improve its own program search mechanisms during the search process. More information on GE can be found at www.grammatical-evolution.org.

The important aspect of the systems discussed above is the goal of managing search bias. This is usually achieved through the use of certain additional (fixed) mechanisms working on top of the pure GP-like search. Explicit bias management, and probabilistic acquisition of search bias based on the history of interactions is then added as a fixed extension of the pure GP model. The next step in revising the model is to provide flexibility with the bias management mechanisms, themselves.
6.5 Beyond neo-Darwinism

Most of the evolutionary computation mechanisms are designed in such a way that all the parameter tuning and expert knowledge about the problem must be provided at the beginning of the search process. During the search process no further adaptation of the overall algorithm is usually available. There is a class of methods that tries to provide much more flexible architectures, in which adaptation of the search occurs during the search process itself. In a way, one can think of these methods as meta-search techniques, where the actual search level primitives can be and are manipulated by the meta-level tuning mechanisms. We will discuss two such examples below: ADATE and PushPop. First however, we discuss the general issues related to the topic of modelling adaptive processes.

6.5.1 Modelling adaptation

Selection-based modelling has proved itself fruitful in many areas of evolutionary research. For practical reasons, the notions of selection and fitness provide useful tools with predictive capabilities. However, the interpretative values of selection-based models (in terms of causality) are questionable. Selection-based models rely heavily on the notion of fitness. In simple terms, fitness represents an external influence or bias that makes a random heuristic search with heuristic function $G$ (Vose, 1999, 3.2, p.9) converge to an attractor $A$. Following Vose’s formalism (Vose, 1999), it is $G$ (and in the special case of the Simple Genetic Algorithm, the fitness/selection/mutation/mixing decomposition of $G$) that guides the process towards a particular attractor. However, for biological systems it is appropriate to think that complex (unknown) evolutionary interactions between all of the interacting entities produce the dynamics with the given attractors. In this view $G$ is just the mathematical result, the final effect, not the cause of the dynamics. As such, models based on the statistically abstract notion of $G$ are not best suited for the task of explaining the detailed dynamics that lead to a particular attractor.

There are alternative models that try to investigate the detailed interactions between all the entities and investigate how those interactions lead to a given attractor. Some of these results are based on the notion of interactions in Boolean systems, artificial chemistries and artificial life. Below, we will look closely into some of the models that do not use directly selection/fitness abstractions.

6.5.2 Evolution without selection

The focus on self-organisation and self-assembly, even though in the majority of cases come from the artificial life and artificial chemistry research perspectives, is occasionally emphasised in the biological research world, too. This is the case with the comprehensive studies conducted by molecular geneticist Lima-de-Faria (de Faria, 1988). In “Evolution without selection”, Lima-de-Faria presents numerous arguments and studies arguing for the role of self-organisation and self-assembly on various levels of organisation and biological development. He argues that the current neo-Darwinist
and selection-based models are not sufficient to model the complex interactions on various levels of organisation.

Despite the growing evidence to the contrary, most mainstream efforts endorse the use of exclusive selection-based models. In (, USA, 2005(), NSF), page 3, we read: “Adaptation requires both that organisms differ in fitness and that those fitness differences be heritable.” Fitness, a simplified modelling approximation and tool for studies of certain coarse-grained statistical regularities, becomes in the statement a first class causal agent. This is clearly at odds with the above shown examples that exhibit complex evolutionary dynamics without the explicit notion of fitness and selection. It may be useful in certain studies to treat fitness and selection as causal forces, but one should not forget that these are ultimately just pure statistical constructs that are derived from deterministic interactions of large collections of components on different levels of evolutionary organisation. Selection/fitness are observable effects of the underlying evolutionary dynamics of complex biological systems. The explanatory capabilities of selection-based models are not applicable or particularly useful for studies of the underlying dependencies and dynamics.

Consider the case of two simple computational self-replicators, \( R_1 \) and \( R_2 \), such that the time to replicate \( R_2 \) is twice as long as the replication time for \( R_1 \). The actual physical cause of this difference may be the length of the program or inefficiency of the self-replication algorithm used (or any other structural or dynamical relationship). Now, when an initial population consisting of 50% of \( R_1 \) and 50% of \( R_2 \) is observed, a simple selection model can be proposed to capture the observed statistical regularity, for example as \( S = < 2/3, 1/3 > \) (S represents the selection probability vector, notations as in (Vose, 1999)). The explanatory power of such a model is however limited to the global statistical effects of the underlying (detailed) dynamics. To understand why the population converges to \( R_1 \), or in other words, why \( R_1 \) is a more efficient self-replicator will not be possible with this simple selection-based model. A detailed model of all the interacting entities with their inder-dependencies is required. The situation is more complex if more than 2 species interact with each other, forming a complex web of parasitic and symbiotic dependencies. In these cases selection-based models rarely lead to good approximation of such a dynamic evolutionary system of the underlying interactions.

Another problem occurs due to finite population sampling, referred to as **genetic drift**. If there are two replicators as before, \( R_1 \) and \( R_2 \), and the replication rate for both is the same, then a selection model is simply unable to model such a system. Any selection model due to the sampling effect, will eventually converge to a single fixed point. The final population will be uniform (either \( R_1 \) or \( R_2 \)). Note, that these sampling effects are inherent in the modelling based on selection-based models. In contrast, any model based on self-replicating entities will not suffer any such statistical limitations. Note also, that the relevance of statistical sampling may or may not have any relevance to the actual phenomena being modelled. It may or may not be just an artifact of the model itself. Studies when self-replication units are used instead of statistical sampling will usually give more accurate models for some classes of complex phenomena.
6.6 Meta-evolution: going beyond traditional models

In (National Science Foundation, NSF) page 3 we read: “Genomes have specific architectures and components that vary dynamically both within and among species. We are beginning to understand that single-gene approaches to understanding genome structure and function are insufficient. Individual gene products are embedded in large-scale interaction networks that represent integrated functional units at the molecular genetic level. Our understanding of the evolutionary dynamics of diversification in genome structures and their associated molecular genetic networks, however, remains limited. [...] we lack theoretical analyses to inform our understanding of how genomes and their associated genetic networks evolve. The revolution in genomics technologies and resources, including whole genome sequences, low-cost sequencing, microarray technologies and computational power, now allow us to begin to address questions surrounding the evolution of genome structure and function.”

This statement suggests the need for theoretical and experimental frameworks suitable for conducting interpretative studies of complex systems based on aggregate networks of interacting entities. Explicit modelling of the dynamics of complex networks of interacting entities is necessary in order to enable research and analysis of complex dynamic genetic networks. A good indication of this can be found in the above-mentioned report: “Complex phenotypic characteristics are determined by interaction of multiple causes. Such characters represent a challenge to developmental evolutionary biologists because there are typically no readily apparent intermediate phenotypes that would be likely to be maintained by selection. The classic example is the vertebrate eye.”

In the following sections we are going to discuss computational mechanisms and state-of-the-art in meta-evolutionary systems. These are contemporary computational models and frameworks that try to model the process of evolving evolvability through recursive application of some evolutionary principles on two levels of the system organisation.

6.6.1 ADATE

Automatic Design of Algorithms Through Evolution, ADATE, is the framework developed by Roland Olsson. Designed and developed as a self-improving GP system (Olsson, 1999, 2002). ADATE has been in development for over a decade (Olsson, 1995). As with other EC models, it is based on the orthodox dogma of genetics, i.e. it uses the fixed mappings between genotype and phenotype levels. However, by placing an emphasis on a neutral theory of evolution (see Section 5.6 for details), and neutral walks, it represents a step outside the realms of rigid models based on modern synthesis and, as such, is a step towards self-adaptability and self-organising evolutionary computation models. To a certain extent the ADATE model tries to mimic contemporary genetics closer than mainstream EC research. The work on the ADATE system continues, and the system is being expanded by new elements (Olsson and Wilcox, 2002), however the full reification of the system is constrained by
the computational complexity and runtime requirements. The re-invention of all the mechanisms within the system, while theoretically sound, cannot be easily demonstrated due to the extremely high computational complexity of the mechanisms used.

The basic assumption of the ADATE system is that the majority of evolutionary changes are due to neutral or almost neutral mutations. Therefore, variability and novelty in species is accomplished by accumulation of a large number of neutral mutations, which are separated by rare brute mutations (as used in ADATE terminology). Neutral walks in genotype spaces are essential for avoiding combinatorial explosions due to complex mutations. Neutral mutations enable evolution to explore fitness plateau and find suitable points where it is easy to reach higher value of fitness. The system is using specially designed rewrite operators that transform the genotype of the program into its equivalent semantic representation with a different syntax. One of the most important properties of the ADATE system is its ability to automatically abstract a concrete body of a function into a parametrised version with one or more variables. Such a template can be later used to reconstruct the original function, by means of substitution (Olsson, 1999).

The EVM and the ADATE systems share certain characteristics. Both of the system architectures are designed according to the principle that in the evolutionary programming framework the user should define a number of primitive operations, from which the system should automatically construct a number of helper functions. However, the ADATE system continues the EC mainstream tradition. This is related to the way in which those primitive functions are constructed. In ADATE this set is required to be small and not necessarily capable of Turing-level computation. The size constraint is purely pragmatic, as with larger sets it becomes exponentially more and more difficult to generate the necessary help functions. The EVM architecture is more flexible and robust in this respect – the key is Turing-capability, which can be constrained if necessary. Therefore apart from abstraction and generalisation, EVM usually starts with redundancy that is subjected to specialisation pressures – this is something the ADATE system does not have. Another feature that the EVM and the ADATE system share in common is the notion of scoping. This is something still not used in mainstream EC models, but as pointed out multiple times by Olsson, scope restrictions are important in automatic search. This is simply due to the size of the combinatorial search space. If many irrelevant functions are available at a given position in a program, combinatorial search for a program instruction becomes exponentially more difficult. Unlike ADATE however, the EVM system facilitates scoping on multiple levels of abstractions. The virtual machine enables customisable and flexible scoping. The EVM allows many concurrent searches on multiple levels to be executed in parallel within the same problem space, with graph-like relationships between different computational elements. In this respect, the ADATE system is similar to contemporary genetics models, and more efficient. The EVM architecture provides more meta-capabilities and is more flexible (but suffers from computational complexity issues).
6.6.2 PushPop

Push is a programming language designed and developed by Lee Spector (Spector and Robinson, 2002). This language is used in a GP system called PushGP. PushGP is the system that shares its design principles with other GP environments, insofar as its genetic operators such as mutation, recombination and selection are applied to the population. The main aim of designing the Push language was to develop a flexible and type-safe language that can be used in ordinary evolutionary computation systems. The language provides advanced genetic programming facilities, such as multiple data types, automatically defined subroutines, control structures, and an interesting form of self-reflection. The Pushpop system is a system that uses Push language and supports a self-adaptive form of evolutionary computation, called in the context of Push: *autoconstructive evolution*.

The language is based on the general notion of stack-based programming (the same as EVM architecture, see Chapter 3 for a discussion on stack vs. register based machines). To provide type-checking, there are several typed operand stacks, which provide operands for specific type-aware instructions. For example integer-based addition instruction will take operands from a special integer stack, whereas floating-point addition instruction would use the floating-point operand stack. The reflective capabilities are provided by the special code stack. This stack can contain the program instructions, and therefore manipulation of the program itself is possible.

The Push language reflective design is similar to the stack-based representation and original assembly language of our EVM model. Spector’s work on hierarchical mechanisms within the evolutionary system PushGP (Spector, 2002) is relevant to the hierarchy in the EVM architecture. However, there are some differences. First, in the EVM system, there is no explicit differentiation between program and data. Data structures and programs are treated uniformly within the EVM framework (no data-program dichotomy). A program can manipulate another program as if it were an ordinary data structure – there are no special instructions or treatment for code vs. data. The other difference is, that EVM is not a typed language. There is only one type of operand stack, and only one type of information (or data) carrier. The EVM is integer-based machine: all data are represented as integers within the EVM architecture. The final difference is the explicit modelling of the hierarchy structures. The main emphasis of the EVM assembly language is the ability to express, create, and manipulate multiple levels of machine organisation. All these features have been already discussed in more detail in Chapter 3. This type of hierarchy is more implicit in the PushGP system, and cannot be subject to evolutionary pressure directly, but only as a side-effect, or indirectly, as a result of the dynamics and statistical properties of the search process.

6.7 Artificial life and Artificial chemistry

Artificial life, (ALife), is a broad term that depicts the area of research concerned with synthetic study of the processes of life. ALife origins can be traced to many diverse fields of research in the natu-
ral sciences, physics, and computer science. ALife works as an umbrella term for techniques and methods of studying different emergent phenomena from various fields. It is often considered as an inter-disciplinary meeting point for efforts and research fields, such as biology, sociology, mathematics, linguistics, physics, computer science, and philosophy.

One of the characteristics of the field is the extensive use of computer programs and computer simulations. Different methods of artificial evolution might be considered subfields of ALife. This includes different evolutionary computation algorithms (GA, GP), swarm intelligence, agent-based models, and cellular automata.

Contemporary ALife systems, historically close to other evolutionary computation models, depart from fixed predefined fitness functions toward open-ended dynamic evolutionary models. Most of the models are based on population genetics. There are some more synthesis-oriented methodologies that explore various aspects of evolutionary dynamics. However, most of such models lack any strict theoretical foundations and are criticised by mainstream evolutionary biologists (for being ad hoc, without a solid analytical and theoretical basis).

One of the subfields of ALife is artificial chemistry. Artificial chemistry is a model of reaction systems, composed generally of the triple \((S, R, A)\): the set \(S\) of all possible molecules, the set \(R\) of \(n\)-ary operations on the molecules in \(S\), and the algorithm \(A\) concerning how to apply rules from \(R\) to the molecules from \(S\). This is a direct derivative or variation on the Chemical Abstract Machine (abbreviated typically as \(\text{cham}\)) model, discussed in detail in Berry and Boudol (1989). Cham has been successfully used as a modelling formalism for other process calculi and process algebras, most notably for Milner’s CCS (Milner, 1989), and Nicola-Hennessy’s TCCS (Nicola and Hennessy, 1987). It is possible to model many asynchronous computational systems within the \(\text{cham}\) formalism. The observations within \(\text{cham}\) can be extended to any other process calculi. In \(\text{cham}\) the state of a system is modelled as solutions consisting of floating data structures (molecules) that can interact with each other according to reaction rules. These data structures can be of any type: primitive (such as numbers and strings), complex objects, or agents. There is a mechanism that “stirs” the solution, allowing for possible contacts between molecules. Note that the solution transformation process is inherently parallel. Any number of reactions can be performed at the same time, assuming that each molecule participates only in a single reaction. Assuming that the data-structures are individual agents, and the interactions are equivalent to reactions in \(\text{cham}\), we can talk about two aspects with respect to autonomy (and indeterminacy):

- interactions are random. They are not pre-ordered or pre-specified by the system design, and
- reaction rules may or may not be followed by the individual agents.\(^3\)

Now, let us consider a particular system (example inspired by Banâtre, Coutant, and Le Metayer (1988)), consisting of \(n\) agents named \(2 \ldots n + 1\) and a reaction rule (interaction) between agents.

\(^3\)Note, that in the original \(\text{cham}\) formalism all the reaction rules must be strictly followed by the system.
such as

if $A_i, A_{i\times j}$ where $i, j \in [2, n + 1]$ then $A_{i\times j}$ annihilates itself.

That means that if two agents meet, and one of the agents is a multiple of the other, the multiple will annihilate itself. From the initial solution of all $n$ agents, after some time, there will be only agents named with prime numbers left. This is assuming both autonomy in the interaction choices and autonomy in the adoption of the general annihilation rule.

The above example demonstrates that in some circumstances, global coherent behaviour can be obtained in systems where autonomy is present on some of the underlying levels of abstraction. However, this is not always the case with all systems. In some systems, autonomy must be restricted for the system to achieve a desirable stable point. For example in the case of *cham*, it is not easy to advise autonomous rules that would lead the system to calculate a factorial. We have discussed this in more detail in the context of our EVM abstract architecture (see Chapter 2).

Artificial chemistry is an important inspiration for the EVM model. The close precursor of the EVM architecture is the binary automata-based artificial chemistry model proposed by Dittrich (Dittrich, Ziegler, and Banzhaf, 1998), which in turn is inspired by the model called typogenetics (Hofstadter, 1979). We also took some inspiration from Tom Ray’s Tierra world. We will briefly discuss these models and the relationship to the EVM system in the following subsections.

The reader can find additional information in Dittrich et al. (1998), which contains an interesting models for evaluating and performing comparative analysis of the population-based artificial evolutionary systems, and the review of the different methods and models that exist within artificial chemistry (Dittrich, Ziegler, and Banzhaf, 2001).

6.7.1 Typogenetics

Typogenetics is a semi-formal system proposed to model the gene-centred view of genetics. It has been successfully applied and used for search of self-replicating DNA strands. It was originally proposed in the book by Hofstadter (1979), and used to model the state of genetics at the end of the 1970s. The author purposefully simplified the genetic model, excluding many chemical aspects of the processes involved. They also simplified the overall interactions between the genes. They have not used any of the new notions from contemporary genetics, such as epigenetics, rnomics, exaptation, and others (discussed in Section 5.3. Typogenetics has been modelled in the form of a formal language with appropriate transformations of the language sequences. Typogenetics was further extensively explored and studied by Morris (1989) and Varetto (1993). More information about it can be found in Andrew Snare’s thesis (Snare, 1999). Typogenetics can model DNA computing, and it has been shown that it is capable of universal computation.

There is some interesting emergent behaviour from some of the models, and the variable instruction set allows the researcher to control the complexity of the overall model. There were some limitations, most notably with the complexity of the search for self-replicators.
The early experiments with the EVM architecture were modelled on simple chemical reaction networks and typogenetics. However, the limitations and the inability to express multi-level dependencies and more complicated reaction chains pushed the EVM design and architecture into different direction. We have augmented the basic mechanism with constructs that enable EVM to create and manipulate hierarchically organised computation and achieve higher complexity levels.

6.7.2 Binary reaction networks

One of the early models of reaction networks was proposed by Stuart Kauffman (Kauffman, 1993). Kauffman conducted extensive studies in the area of random Boolean networks, which could be classified today as proper artificial chemistry models. Kauffman investigated some of the properties of these networks: pattern generation, self-organisation and autocatalysis. He also investigated different fitness landscapes and their influence on the overall dynamics of the network. Random Boolean networks are directed graphs, with a fixed or a random number of interconnections leading from and to individual nodes. The node performs a randomly assigned Boolean operation on the inputs, and propagates the outcome of the operation to all its outputs. This simple model can exhibit most of the evolutionary computation phenomena in the domain of Boolean functions. It can be used to study different fitness landscapes and dynamics of the overall pattern formation.

The second important contribution in the realm of Boolean reaction systems is work by a German research group, led by Wolfgang Banzhaf, Peter Dittrich and Jens Ziegler (Dittrich et al., 1998). They have conducted experiments with binary strings that reacted with one another in a single container. There are several different possible reaction rules, and the main aim of these models is usually the investigation of catalytic reaction networks, auto-catalysis, and self-replication. A thorough review of the reaction network and artificial chemistries can found in Dittrich et al. (2001).

In the early stages of the work on the EVM architecture, the design was based on the binary computer design of Boolean reaction nets (Dittrich et al., 1998). The basic idea of coding the processor as a binary string and input-output data as binary strings is derived from those earlier reaction models. However, for the sake of flexibility and possibility of expanding instruction sets, we extended the binary strings into integer strings, and designed the current model of the EVM integer-based coding. However, the EVM computing model still shares some of its properties with reaction networks, and it is well suited for some experiments within the domain of artificial chemistry and reaction networks. Some of the experiments will be discussed in details in Chapter 9.

6.7.3 Tierra and Avida

Tierra is one of the first computer simulations of a digital ecosystem, designed and implemented by Thomas S. Ray (Ray, 1991a,b, 1995). In the Tierra system, there is a population of programs (organisms) subjected to some evolutionary operators, such as mating, crossover and random mutations. However, unlike other evolutionary models discussed earlier, there is no explicitly defined
fitness function. Instead, the fitness of the individual depends on its ability to maintain itself, and self-replicate. This enables Tierra to be used in experiments with open-ended evolutionary processes and to investigate systems with the dynamics of feedback between evolutionary and ecological processes. Programs in Tierra are sequences of machine instructions, which compete for processor time and for memory. Programs can overlap, and interesting host-parasite co-evolutionary models have been investigated.

Because there is no explicit fitness function, Tierra, on its own, cannot be used as an optimisation algorithm. It is designed purely so that the dynamics and emerging phenomena in the evolutionary and co-evolutionary framework can be investigated. All programs (organisms) are executed by the shared single processor, and they occupy a global memory space.

Avida is another Tierra-like simulation framework, which derives directly from Tierra. The main difference between Avida and Tierra is that in Avida, every program lives in its own protected region of memory, and is executed by its own virtual CPU. By default, other digital organisms cannot access this memory space, neither for reading nor for writing, and cannot execute code that is not in their own memory space. Another important difference between Tierra and Avida is the use of explicit selective pressures. Additionally to the competition for resources, there is extra selective pressure that enables the evolutionary processes of Avida to be used for optimisation purposes.

The EVM architecture takes some of its inspiration from the original Tierra system. In the EVM architecture the stress is placed on the inflow of information, or signals, from the outside environment. This was not present in the original Tierra system. There was no actual information growth, or algorithmic complexity growth in Tierra. The EVM architecture overcomes that limitation.

6.8 Evolvability

One of the important aspects of the research conducted by Thomas S. Ray was a comparative study of different patterns of evolution with four slightly different machine languages (Ray, 1994). Ray observed that two out of the four different languages showed significantly greater capability to adapt than the other two. The two less evolvable languages showed a relatively strict patterns of gradualism of evolution, whereas the other two showed abrupt jumps in the evolutionary process; clear punctuations. Of the two showing punctuations, one demonstrated gradual evolution between the punctuations, while the other showed strict stasis between the punctuations (Ray, 1994).

It seems important to distinguish between environmental constraints (including material constraints emanating from the qualities of components of a system) and self-imposed limitations associated with the particular path taken as a dynamical system unfolds through time. In other words, I see some information being generated by the dynamics of a system, much of which can emerge from the interaction between a system and the constraints of its environment. I have come to this view largely by considering
the process of biological development. For example, I have come to the conclusion that
the genome is far from a blueprint of a phenotype, although it is more than a static list
of building parts. I see the genome as containing a small fraction of the information
ultimately represented by an adult organism, and I think that most of that information
is generated internally to the system as a consequence of the interaction between the
genome and its environment.

Guy A. Hoelzer, hoelzer@unr.edu, private communication

The EVM architecture builds on these results and explores the potential of evolving evolvable
languages, i.e. languages that exhibit certain predefined set of qualities. The strength of the EVM
architecture lies in its meta-evolutionary approach. The idea is to explore and exploit those dynamical
regimes, that lead to desirable evolutionary properties. Ray has demonstrated that the dynamics of
the evolutionary process depend on the structure of the underlying language. Not only on the lan-
guage itself, but also on the set of possible genetic operations, and the available interactions between
different language elements. Therefore, this is rather difficult to conduct EC research that abstracts
from particular modelling assumptions (or language). Not surprisingly, there is a limited amount of
research conducted within the area of evolvability. For many evolutionary computation researchers
it is still a black art, acquired by a trial-and-error approach with different parameters, hand tuning,
and following intuition. Some of the languages used in particular applications are evolvable, while
others are not. Some of the languages are only evolvable in particular circumstances, and do not work
well in all the cases. There is no established theory of how to design and enable enhancements of a
particular language to make it highly evolvable. Apart from Ray’s work, only the Push programming
language was designed with the concept of evolvability as a primary architectural design objective.

EC models are a good source of information and research in the area of evolvability. However,
there are some open questions and issues that need to be addressed. EC models usually seek a method
to find a single global optimum in the pre-defined landscape that is shaped by an explicit fitness
function. ALife models go a step further, and allow the search process to be somewhat open-ended,
with an unknown fitness function. However, it is important to realise that in the typical ALife system,
such as Tierra for example, even though the fitness function is unknown, it is usually fixed, too.
Many ALife models such as Tierra allow experimentation with more dynamic setups, that is, the
fitness landscape can be shaped on-the-fly by the co-evolutionary processes. This is why the synthetic
approach to life is generally a good and successful approach to the issues of evolvability.

Within the realms of the EVM system, the evolving cell (see Chapter 3) tries not to get trapped
in a local optimum. There is no single objective function, but various different sources of “reward”
or replication. Thus instead of performing simple hill-climbing, the cell continuously tries to re-
shape its own fitness landscape, through various exploration techniques. These explorations are then
augmented with hill climbing so as to test if the achieved niche offers hill-climbing benefits, i.e. to
climb up in the new fitness landscape from the currently occupied plateau. The simple idea is that
biological evolution can create continuously new evolutionary pathways in co-evolving species. This is achieved by creating new species and new niches, continuously, at the same time. There is no real notion of a fixed niche and a fixed selection pressure that is given from outside (as in most EC models). The actual selection pressures are always local, and this is what the EVM architecture models and investigates. The new organisational levels in the EVM model are the key to understanding the open-ended evolutionary process. Because new levels can always be constructed out of the existing genetic makeup, the evolutionary processes will never settle in the state of optima, but will continuously create higher and higher levels of organisation (see Chapter 7). We believe that this property is essential in modelling evolutionary dynamics. The EVM system tries to facilitate the investigation of the process of diversification of new evolutionary forms and the creation of new levels of organisation.

### 6.9 Hierarchy in Evolutionary Computation

Traditional evolutionary program-generation techniques, like ALife or GP with tree-like or linear chromosomes, use human-designed virtual machines with particular properties. The machine is designed to fit into the widest possible class of different problems in a given problem area. Usually, that decision is outside of the requirements for evolvability. Later, a set of “benchmark” problems is tested on a given virtual machine, and inferences are drawn concerning its suitability for a specific class of problems. This methodology works within narrow domains and can be successfully applied to many scientific and engineering problems. Nevertheless, it is stochastic “guesswork”. We propose to unify existing representations under a single common framework based on the notion of hierarchical virtual machines. Such unification provides a more systematic approach which we believe leads to better automation of the search for an optimal virtual machine hierarchy for a given problem.

As noted by different authors (Koza, 1995; Rosca, 1997), all program-generating algorithms use some sort of hierarchical problem decomposition, usually implicitly. Otherwise, the search space would quickly become unmanageable, due to a fast growing problem complexity (Koza, 1995; Schmidhuber, 1999; Spector, 2002).

The idea is to introduce self-referential and recursion manipulation capabilities to the base level of the human-designed virtual machine. For example, in the PushGP system (Spector and Robinson, 2002), the primitive instruction set contains some specific code-manipulation functions that provide recursion and iteration mechanisms. Similar techniques are being used in slightly modified forms in the work of other researchers (Schmidhuber, 1999; Olsson and Wilcox, 2002). The main characteristic of such modelling techniques is that effectively all the abstractions levels can be collapsed to a single self-referential base level. The hierarchy levels are controlled by the initial system settings, and they are automatically generated by the mechanisms of the search process. This can improve the search process for some classes of problems. However, our objective, which is the analysis of interactions between the hierarchy levels, is not achieved due to the above-mentioned level-collapse. All the intermediate virtual levels are created automatically and implicitly. What we want is an environment
where the levels can be retained and used explicitly by the search process.

### 6.9.1 Evolving recursive virtual machines

The field of EC is mainly based on experimentation: a simple trial-and-error approach. In light of all the advances in theoretical computer science and given the conceptual framework of recursive virtual machines, it is now possible to introduce a more systematic approach. Each different evolutionary system is an example of a virtual machine, each language is an example of a different search space, and each system is an example of the interplay between different aspects of the hierarchical organisation.

Probably one of the closest existing systems using the concept of a tower of virtual machines with a hierarchy of virtual machines is the grammatical evolution system (Ryan et al., 1998). In this system, a top-level search is performed on strings of integers. A string containing integers is fed into a particular machine to produce a computer program coded in a particular language as output. This code is then fed as input to yet another machine, which in turn returns a final result. Each of the levels is relative to the level below it; this relativity means that the same top-level string of integers will produce a completely different result when used in combination with another machine. The top-level machine accepting the strings of integers is designed in such a way that it can “plug-in” to any possible second-level machine, and the model will still work. This is a human designed feature, but it is inspired by many naturally occurring phenomena. The multiple levels of indirect influences seems to be the most powerful mechanism at work here.

Instead of designing such machines and all the indirection levels by hand, it is believed that this process can be automated. The virtual machine suitable for a particular class of problems can be discovered automatically. In fact, as shown in the Example in the Boolean domain section (below, Section 6.9.3), some of the virtual machines are simple enough that an exhaustive search can be used to discover them.

### 6.9.2 Seeds and solution growing

Let us take a grammatical evolution system (Ryan et al., 1998) as an example of a solution growing-concept. The solution for a problem at hand is effectively a proper hierarchy of machines (in this case a BNF-encoded language grammar) and a string of integers as a symbolically encoded solution, which we refer to as a seed. In the case of a grammatical evolution system, the hierarchy of machines is designed by a human programmer before the search for the proper seed is started. However, the hierarchy of machines needs to be discovered, as well, thus it is best to treat it as part of the solution itself.

In general, the solution to the problem (finding a computer program) is a hierarchy of machines, together with the seed. The actual computer program is then generated by feeding the seed through the system. In the case of grammatical evolution, speaking informally, the generation process is (in
order): feeding the string of integers, generating the program listing, running the program for the
given input and then obtaining the final solution. The given input in this case depends on the “outer-
level” virtual machine.

It is, however, possible to change or modify the machine hierarchy just before generating the
computer program. If the hierarchy of machines, their connections and the initial states are subject
to change, we refer to the process of generating a final solution as solution growing. In the case
of searching for code one can use the term code growing instead. It is possible, by varying the
hierarchy of machines, to grow a valid solution from the same seed for a certain variation of the
original problem. By simple re-mapping, one can achieve exactly the same result by varying the
structure of the seed itself. This opens a new window of opportunities not yet used by the automatic
code generation techniques. Again, it is a commonly occurring phenomenon in nature.

Formally the idea of code growing is based on the notions of bootstrapping and self-application.
This is analogous to more traditional compiler/interpreter bootstrapping and self-application (Jones,
1997).

6.9.3 n-XOR problem

The n-XOR problem is used here to demonstrate a recursive and hierarchical nature of some of the
problems and how EVM-like system can take advantage of this property. Boolean algebra is widely
used for different benchmarks in EC (Wong and Leung, 1996; Gathercole and Ross, 1997; Chellapilla,
1998). Within the context of evolutionary computation, Langdon and Poli present an extended sum-
mary of different problems and functions in Boolean algebra (Langdon and Poli, 1999). We will
investigate one of the more difficult problems: generating a Boolean expression for an odd-n-parity
problem, referred to here as the n-XOR function. Note, however, that in our set-up for this problem,
we literally mean n-XOR, where \( n \) is the actual input parameter. Traditionally, researchers tackle a
particular instance of this problem, for a given \( n \). Our problem is, in a sense, a meta n-XOR search.
The choice of this particular function is based on the observation that this is the most difficult of all \( 2^n \)
Boolean functions, i.e. the actual expression in conjunctive or disjunctive normal form is the longest.
For all other functions the expression is shorter than for n-XOR.

The final solution expression for a given \( n \) will be constrained to use only some predefined
Boolean operators. n-XOR is a symmetrical Boolean function which returns 1 if and only if the
number of activated inputs is odd, and 0 otherwise. The complementary function is called even-n-
parity. In both cases the usual logical operations to build an n-XOR expression are binary XOR
(exclusive OR) and EQ (equality). However to make the problem more difficult, we will constrain
the operator set to be two 2-variable operations: AND and OR, and one 1-variable operation: NOT.
We also require the solution to be of minimal size, in accordance with the aim of traditional boolean
function minimisation techniques. The size is the length of the expression, in other words, the number
of nodes in the expression tree representing the n-XOR function.

180
Again, note that the prior results discussed below are for a different problem, where only a single instance of the n-XOR truth table is available for fitness evaluation, not the whole class, as in our case. The class is usually to be discovered automatically based on a single case example from the class. Poli and Page report finding the solutions for fixed \( n \) up to 22 (Poli and Page, 2000); however the results are far from being of minimal size. Note, that in their experiments the XOR operator is available, and yet the result for the odd-6-parity problem is almost 5 times longer than the minimal size expression. There is no known publication which reports the discovery of a minimal size solution for this problem, at all, even with relatively small values of the parameter \( n, \ n > 5 \). The best and the closest to the ideal solution is the one informally reported by Roland Olsson. Olsson used his ADATE system (Olsson and Wilcox, 2002), and his solution to the same problem (2-variable AND/OR and 1-variable NOT only) for 9-XOR expressions has an impressive 265 nodes, which is only slightly larger than what we believe to be the shortest expression (233 nodes).

Why is that so? Why is the problem “unsolvable” by traditional EC methods? As shown in (Langdon and Poli, 1998), the fitness landscape (the search space) for a GP-like system is extremely rugged and does not allow easy ascent toward optimal solutions. The search space must thus be “transformed” by the GP machine into a more manageable one, which effectively causes code bloat. The code bloat for large \( n \) occurs extremely rapidly, and only methods with proper parsimony pressures can cope with it to a certain extent.

According to (Furst, Saxe, and Sipser, 1984), the parity functions cannot be represented by polynomial-size constant-depth AND-OR-NOT expressions, which means that the size of the minimal expression grows with \( n \) faster than any polynomial. With 2-arity operations it is also impossible to have a constant-depth solution, which means that the expression depth increases. This makes it extremely difficult for tree search techniques, GP being a perfect example, where nodes closer to the root are difficult (or impossible) to change. The search space simply becomes unmanageable for larger values of the parameter \( n \).

What can we say about appropriate virtual machines designed to solve the n-XOR problem? We know that traditional GP-like code generation will simply fail for a large number of variables but can be successfully applied to solve it for small values of \( n \) (Langdon and Poli, 1998; Poli and Page, 2000). The lower bound for the expression size has been proved to be exponential by (Håstad, 1989). It seems evident that we have to escape to recursion, otherwise no matter what method we use, the growing search space will become unmanageable for large values of \( n \). We have the interplay of two important aspects: the language in which the expression is expressed and the machine which accepts that language. We also know that our hierarchy of machines has to be dynamic and it has to grow as \( n \) grows.
6.9.4 n-XOR hierarchy

Our solution consists of a tower of simple substitution systems. Each machine in the tower accepts the input stream, and for each of the symbols from the input stream, it performs a predefined symbol substitution. Each machine can have a different set of rules, or can have exactly the same rules. In the latter case we deal with the same set of rules recursively applied to itself a multiple number of times (self-application). We write the substitution rules as follows: on the left-hand side of the expression we write the individual symbol tuple, and on the right-hand side we write the result of the substitution as a tuple. All the inputs and outputs are sequential.

Let us assume that the input stream consists of a sequence of variable names. For a 1-XOR problem we would have a single-variable input stream, say \(<x_1>\). For 2-XOR we would have two variables: \(<x_1, x_2>\), for 3-XOR three variables: \(<x_1, x_2, x_3>\), and so on. In the general case of n-XOR we have \(<x_1, x_2, \ldots, x_n>\). Of course the final expected output is the actual expression representing the n-XOR problem, using only the predefined operators.

In the search for a suitable set of rules, we start with the divide and conquer principle, i.e. we try to split our problem into two problems, solve each of them separately, and then combine the results. To do so, first we use an exhaustive search and try to build a shortest solution for the two-variable case. Knowing how to solve the 2-XOR case will allow us to “combine” two sub-solutions later in the process.

Let us estimate first the search space for 2-XOR. We have \(t = 2\), where \(t\) denotes the number of terminals (variables \(x_1\) and \(x_2\)), and \(f = 3\), where \(f\) denotes the number of different operators, which for simplicity we assume to always have two arguments. Thus, according to the formula for the number of different expression trees for a given tree size \(l\) (Koza, 1992):

\[
\frac{t^{(l+1)/2} \cdot f^{(l-1)/2} \cdot (l-1)!}{((l+1)/2)!((l-1)/2)!}
\]  \hspace{1cm} (6.1)

we have no more than 55000 different trees of size 9, which yields an estimate on all the trees up to size 9 to be not more than half a million. Our search space for the 2-XOR problem is that big, because the minimal expression for 2-XOR is of length 9. To find a solution we do not need any particular technique, we can use an exhaustive search here, i.e. iterate through all half a million possible solutions.

(Note: for simplicity, we use the following notation conventions: AND and OR are 2-arity Boolean functions, represented by ‘.’ (as in arithmetical multiplication) and ‘+’ respectively. To simplify the expressions, we omit the explicit representation of the AND operator and assume its operation when two operands are juxtaposed. In equations 6.2 and 6.3 all operators are written explicitly to ease counting of the expression size. NOT is a 1-arity operator which is represented as an over-line (e.g. \(\overline{x_1}\) for \(\text{NOT } x_1\)).)

The exhaustive search yields two solutions (assuming the variables are \(<x_1, x_2>\)):

\[
\overline{x_1} \cdot x_2 + x_1 \cdot \overline{x_2}
\]  \hspace{1cm} (6.2)

182
If we rewrite the first of the found solutions as a rule for our substitution system, we will have:

\[ \langle x_1, x_2 \rangle \rightarrow \langle x_1 x_2 + x_1 x_2 \rangle \] (6.4)

We need also note that the 1-XOR solution is the actual single variable, itself, because the 1-XOR is effectively an identity transformation. We can write that as a rule for halting the substitutions (note the halting symbol \( \perp \)):

\[ \langle x_1 \rangle \rightarrow \langle x_1, \perp \rangle \] (6.5)

With the above two basic rules, found via the exhaustive search, we have built a machine for solving 1-XOR and 2-XOR problems. For 1-XOR it simply halts, presenting the result on the output; for 2-XOR it performs the substitution according to the rule and presents the results to the output.

Now, imagine that we have input that is more than two variables and that our machine can take two input variables that appear sequentially and perform the appropriate substitution to produce an intermediate output, which is then passed as input to another machine. Let us assume that all the machines use exactly the same two rules. For the n-XOR problem we would then have a tower of \((n/2) + 1\) such machines.

Let us refer to \( M_1 \) as a substitution machine based on the rules above. As an example, if we feed \( M_1 \) with two input symbols, the first being \( \overline{x_1} x_2 + x_1 \overline{x_2} \) and the second being \( x_3 \), we will get as a result:

\[ \overline{x_1} x_2 + x_1 \overline{x_2} x_3 + (\overline{x_1} x_2 + x_1 \overline{x_2}) x_3 \] (6.6)

If \( N_1 \) denotes the machine based on the rule from Equation 6.3, and with the same input strings as above, we will get:

\[ (\overline{x_1} x_2 + x_1 \overline{x_2}) x_3 + (\overline{x_1} x_2 + x_1 \overline{x_2} + x_3) \] (6.7)

It takes an exhaustive search to show that there is no better set of rules on each single level than the one we have discovered originally. There is no hidden or implicit relationship between our dynamically created levels. Because each of the machines in the tower has exactly the same set of rules, we may connect the output of the first machine with its own input, and we will have the case of recursive self-application. The output goes back to the input as feedback. There are effectively two cases now. The first is where there is more than one input string, in which case the first two are read from the input. Then the substituted formula according to the appropriate rule is prepared, and the result goes back at the end of the input stream. The second case is if there is only one input string left, in this case this is the solution, and the rule 6.5 is applied. The effective solution is in a form of a basic recursive function, where the number of recursions depends on the length of the original input. Remember, the original input is the enumeration of all the variables.
The total size of the expression for our substitution system is given by the recursive formula:

\[ S(1) = 1 \]
\[ S(n) = 2S(n\%2) + 2S(n\%2 + n \ mod \ 2) + 5 \]  

(6.8)

where we have used the notation: \( a\%b = \text{floor}(\frac{a}{b}) \) and \( a \ mod \ b = a \mod b = ((\frac{a}{b}) - (a\%b))b \)

Are such simple solutions like the one for n-XOR a common feature for a large class of problems, or is it just one lucky example? The investigation and classification of different problem classes with respect to the hierarchical decomposition seems to be a key to answering this question. From what we have observed so far, it seems to be a common property for all symmetric Boolean functions. The investigation of more problems in Boolean algebra can tell us more about the general properties of this domain.

6.10 The EVMA as an ALife model

The basic idea of the EVM Universe is the concept of loosely connected network (graph-like or grid-like) of independent, asynchronously executing EVM cells. A single EVM cell is a unit of computation consisting of the EVM program (or programs) running on the EVM virtual machine.

The **EVM cellular system** is characterised by many locally interacting components (EVM cells). Such a system effectively performs a parallel, decentralised, highly redundant computation, e.g. (Sipper, 1999). The overall architecture of the EVM system consists of individual EVM machines (including a list of primitive or complex individual programs), and of the external environment that manages the individual machines, executes programs, schedules tasks, and collects statistics on the use of individual programs.

There are many possible implementations of the actual environmental mechanisms for the EVM architecture. The strength of the EVM architecture is that the users are not locked into a rigid design, but instead, there are many possible implementations of the management layer, all working together on the same network of EVM hosts and EVM cells.

A cell is the fundamental component of our system. All cells run asynchronously and in parallel on multiple EVM hosts. Using the current implementation, each single host is potentially composed of up to \( 2^{32} \) cells indexed by a single 32-bit integer. All individual cells react to local interactions with their neighbours and with the environment. Note that the notion of **locality** is strictly relative, and does not need to follow any spatial organisation. From a machine learning perspective, the goal of each of the cells is to solve one of the tasks available in the environment, whereas from an artificial life perspective, the cell aims at collecting enough food to survive. The cell without enough food is recycled, which means the content is substituted with a new content (old program is replaced with a new program).

In the EVM framework the knowledge is encapsulated within individual cells’ programs. To take advantage of the information stored in other cells, each cell can execute other cells’ programs. When
a cell gains a reward, it will share it with other cells used to compute the solution. All of them benefit from their associations and dependencies. Using a biological analogy, symbiotic relationships will appear between programs. This ability to access neighbours’ programs has opened the door to complex hierarchical organization and self-assembly. Cells are now able to collaborate to solve complex problems. The result is a set of simple, independent, and autonomous problem solvers, working together in a dynamic environment, solving multiple tasks, and dynamically adapting to changing requirements. The EVM architecture provides a self-organising, self-adaptable cellular system for multitask learning.

6.11 Comparison with Swarm and RePast systems

Swarm Langton, Minar, and Burkhart (1995); Minar (1996) and RePast Collier (1999) systems belong to multi-agent simulation toolkits, and inherently operate on higher level than EVM itself. Multi-agent systems do not represent computing architectures or virtual machines in strict sense of these concepts. The EVMA fits in the research area of low-level, traditional VM models, such as JVM or .Net or Forth VM (from which it derives its inspirations). The proposed EVM architecture enhances the computational model by unique instructions that allow much more direct manipulation of the underlying computational machine itself, including, the complete re-instantiation of the instruction set. The instruction counter, the stack and the execution frames can be freely manipulated as any other data-structure – this provides a unique self-reflecting computational architecture. Based on that architecture higher-level abstractions and multi-agent programming paradigm can be built, but direct comparisons to Avida, Tierra, Swarm or RePast systems are difficult to justify or even conduct in the context of EVM studies.

The difference of EVM system as compared to Swarm or RePast is on conceptual, architectural and implementation levels. In the same way as Turing Machine is different to lambda-calculus and recursive functions, in the same way EVM is different from any other complex system modelling paradigm. They all express the same concepts and they are the same in terms of their computational and expressive power. Yet, they all are different. The differences have been outlined in Chapter 2 and 3 in more details.
Chapter 7

Autopoietic hypercycles
Life is not made of atoms, it is merely built out of them. What life is actually “made of” is cycles of cause and effect, loops of causal flow. These phenomena are just as real as atoms – perhaps even more real. If anything, the entire universe is actually made from events, of which atoms are merely some of the consequences.

(Grand, 2000, page 6)

7.1 Computational evolution

In the previous chapter we have reviewed some of the existing models within the field of Evolutionary Computation (EC) and Artificial Life (ALife). Existing EC techniques are applicable to a wide range of modelling, simulations and problem solving scenarios. EC, however, is not trying to capture or model biological evolution or evolutionary systems as such. It is rather an attempt to use inspiration from biological evolutionary theory to solve combinatorial problems. In this chapter we will look into more universal models of evolution and computational evolution, which instead of solving combinatorial problems, try to model various phenomena of biological evolution.

The theoretical approach to biology is carried out mostly by theoretical evolutionary biology. It is nevertheless influenced by results in different fields, including chaos theory, physics, mathematics, and also computer science. Building abstract, information-centric models and analysing properties of such models provides valuable insights into the fundamental laws and regularities of different complex biological and physical phenomena. These provide insights into the nature of complex evolutionary systems. It is believed by some contemporary scientists, that the information-centred approach is a correct, if not the only, possible path to pursue research and make progress in the fields of theoretical and computational biology (Orgel, 1973; Chaitin, 1979). Others, on the other hand, argue that this, in principle, will never provide ultimate understanding of the process of biological life (e.g., (Boden, 2000)), due to its intrinsic intractability in the mathematical sense.

It is not our intention to argue one way or the other, but rather pursue the research in the information-centric domain so that better understanding of the problem can be obtained. There is an emerging area of research concerned with new formulations of theory of information and information processing and of computation, to bring the computation theory and biological theory closer together. These efforts try to unify formal and rigorous information-centric approaches with less formal natural and life sciences by providing a sufficiently general framework that could accommodate both paradigms in a single unified model.

The main efforts within EC research is placed on information-centric methods that mirror the process of the Darwinian theory of random mutations and natural selection. This is visible in well-established computational models of evolutionary processes, such as Genetic Algorithms (GA) (Vose, 1994), Genetic Programming (GP) (Koza, 1992), Evolutionary Programming (EP) (Fogel, 1998), and their variations such as assorted Artificial Life systems. However, simple single-layer evolutionary
systems based on random mutation and selection have been shown to be insufficient (in principle) to produce an open-ended uniform evolutionary process with potential multiple levels of translation. For details see work of Eigen and Schuster (1979) and also work of Wright (1931).

EVMA, along with the theory of hypercycles (Eigen and Schuster, 1979) and autopoiesis (Maturana and Varela, 1980), is our attempt to provide this unified information- and computation-centric paradigm for study and modelling of the processes of life, adaptation, and evolution. The intrinsic properties of hypercycles allow them to evolve into higher levels of complexity analogous to multi-level or hierarchical evolutionary processes. Our main focus is on such open-ended, hierarchically multiple-level structures of self-organising and self-maintaining ensembles – evolving a hierarchy of complexity. We use the basic concepts from the information-centred approach, and we augment them with concepts taken from the theory of hypercycles and autopoiesis. Throughout this work we use two basic notions of information as introduced by Shannon (Shannon and Weaver, 1949) and in Kolmogorov-Solomonoff-Chaitin algorithmic information theory (Li and Vitányi, 1997; Papadimitriou, 1994). We refer to the former notion as information and to the latter as algorithmic information. Due to its nature, the EVMA is inherently appropriate for discussion within the context of algorithmic information and we use and apply some of the intuitions from the algorithmic complexity (Section 4.3).

7.2 Information-centred models of evolution

In this section we summarise briefly what have been already said in previous chapters in the context of information-centred models of evolution. There has been many attempts to define life, complexity, organism, organism boundaries and information content (Schrödinger, 1945; Prigogine and Stengers, 1980; Margulis and Sagan, 1995). There have also been an extensive amount of research conducted on the notion of mechanical self-replication, dating back to the original work of von Neumann (Sipper, 1998). Some authors have attempted to give rigorous quantitative definitions of these concepts in a formal deductive form (von Neumann and Burks, 1966; Chaitin, 1979; Gecow, 1975a). Interestingly, authors coming independently from different sets of basic definitions and assumptions reached similar conclusions (e.g. (Chaitin, 1979) and (Gecow, 1983b)). According to theoretical and experimental work of most authors, the process of improvement in individuals and ensemble growth is best accomplished by carrying along all or almost all of the previously developed structures, while new pieces of an ensemble structure are being added (Simon, 1968). Simulations and statistical analysis within different areas of Artificial Life experimentally confirm the same theoretical conclusions. This indicates that it is more effective to add and patch an existing structure, instead of rebuilding and restructuring it from the core. Often, a simple patch on the boundary of an organism is enough to provide a competitive advantage or allow adaptation to new slightly different environmental conditions.

Some of the ideas within the EVMA have been inspired by the work of Andrzej Gecow, who attempted to give rigorous quantitative definitions for concepts, such as organism, living ensemble,
life and evolution, in a formal deductive form (Gecow, 1975b; Gecow and Hoffman, 1983; Gecow, 1986). The EVMA uses a similar notion of relative organism boundaries, but departs from Gecow’s model on many other assumptions, especially in terms of capabilities of computational cells within EVMA.

The roots of the EVMA model can be traced back to the original work of Turing (Turing, 1948) and John von Neumann (von Neumann, 1963; von Neumann and Burks, 1966). Turing proposed a first formal model of u-machines (Turing, 1948) that captured the notions of reinforcement learning. Von Neumann submitted that a precise mathematical definition must be given to a basic biological theory. The work of von Neumann has been, most noticeably, pursued and extended by Gregory Chaitin (Chaitin, 1970, 1979). Similarly to von Neumann and Chaitin’s models, EVMA is based on a discrete state model universe, i.e. automata space. Note, however, that we try to capture the notion of an open system, where our architecture is capable of interacting with the real world. A formal definition of information used throughout Gecow’s model is defined as an ensemble (as it was originally proposed in (Shannon and Weaver, 1949)) rather than as algorithmic settings (as in (Chaitin, 1979)).

7.3 Properties of computational evolution

In this section, we discuss the properties that are desirable in a generic computational evolutionary system. These can be achieved by certain architectural decisions and system architecture. There are also certain properties that the self-adaptive and self-organising software system may spontaneously exhibit, which can be desirable or not. Some of the properties listed below facilitate effective processes to help and guide evolutionary mechanisms. Some are essential in terms of proper evolutionary dynamics. And some are inspired or motivated by the current mainstream models.

7.3.1 Split and splice

It is desirable that different individual functional units be freely manipulable. It means that one can put different components together and split them apart, always producing valid functional units within the system. This property is referred to as closure in GP literature. Note that we not only require closure to mean that combining any valid programs will produce a valid program; we also want that any division or partition of a valid program also produces valid programs.

7.3.2 Cyclic behaviour

All individual components of the software system must carry out their duties as part of a cycle. That means that the functionality is organised in such a way that tasks are repeated over and over again. Hence tuning, self-organisation, and adaptability can take place. If a given task is designed to be performed only once, there is no room for improvement, as a component has only one single
opportunities to perform. Continuous cycles and repetitions are needed for effective feedback loops to form. In Tierra-like ALife systems this is achieved by executing instruction sequences from the beginning, after the execution of the last program instruction. In GP-like systems this property is not directly exhibited, but is inherent in the overall workings of the GP algorithm, and the improvement is hard-coded into the algorithm itself. There are some meta-GP systems where the improvement mechanisms are subjected to evolutionary pressures, too, but these are beyond the scope of normal GP systems.

7.3.3 Many agents on many levels

There are benefits from having many independent interactive components acting on many different levels. Reflection and recursion in components can facilitate shorter and more robust solutions to given tasks performed by components of the system. Some components will just perform tasks, some will monitor others performing tasks and provide necessary feedback for improvement, and others will improve the “improvers”, and so on. Also, programs from different levels can be re-used on other levels.

7.3.4 Hypothesis testing

There must be a possibility to test hypothesis and test responses of particular changes in the system without the need of stopping or restarting the entire search system. For example in evolutionary computation, there are many generated solutions; all are tested, and the best performing ones are selected for further iterations. It is, however, more efficient to fine-tune some of the existing solutions by running simple (non-selective) executions of the tests in a sandbox-like environment, so individual solutions may test themselves before the real selective pressure could weed them out. This can allow mechanisms such as the Baldwin effect to take place and allow the evolution of learning to take place too. For example, this is how animals learn and master various skills through play during their childhood. Later, these skills are used and employed for their survival.

7.3.5 External information flow

External information flow simply means that the system has to interact with the outside environment. A software system that does not exchange any information with the environment outside the system itself cannot in any case evolve into a more complex system than the original one\(^1\). Such a system can only transform and refine itself, without introducing any new information. New information must be provided from the outside environment, otherwise, the whole system will lack the ability to produce any novelty. More detailed discussion on contemporary computing systems and the concept of randomness has been provided in Section 4.4.

\(^1\)This is due to algorithmic information theory and the fact that any program’s output can be reduced to the shortest program capable of generating this output.
7.3.6 The sustainability condition

Sustainability is often considered a crucial element of any model of a living system. It is stressed in both hypercycle and autopoietic theories and plays a central role in many artificial life systems. It plays a central role in the EVM architecture, too. In the long run, only structures that can sustain themselves can continue to exist. This simply leads to the fact that the meta-purpose\(^2\) of every EVM program is to exist. Formal introduction of the to-exist condition, however, would required a clear definition of identity criterion. Note that trying to use unique identities for higher-order sustainable systems is not appropriate in the same sense as it is valid for flat autocatalytic structures. Hence, the sustainability condition, as such, cannot really be formally introduced into the EVM architecture. In our experimental studies we show certain patterns of organisation to persist, and we often use the metaphorical description such as: a certain structure sustains itself. However, we do not explicitly model, nor maintain, any of the mechanisms or goals of individual cells. It can be considered to be an emergent property of the dynamics of the running system.

Locally, in a given frame of reference, it makes sense to discuss sustainability and identities, and we often use those terms when explaining the behaviour of our system. Overall however, we search for better terminology and for a conceptual framework better suited for discussions of living systems (hypercyclic autopoietic systems). Sustainability, then, falls into a type of operational definition that is used throughout this work.

For EVMA cells, sustaining the existence and maintaining identity is different than in other computational systems. For example in Tierra or Avida systems, the program that is capable of replicating itself and spreading in the digital ecosystem is actually the same program, with the same instructions. There is a direct relationship between what the program executes and what is being sustained. In our case, on many occasion there is no direct link – certain structure are sustained by means of other lower level, in many cases chaotic, processes. It would be difficult (or inappropriate) to draw clear causal links. It is nevertheless important and valuable to investigate these non-direct relationships, and why certain structures come to being and are being maintained within the dynamics of a complex software system.

7.4 Darwinian systems

In preceding sections, we have specified a list of properties that a theoretical model of evolutionary process should have in order to be capable of modelling open-ended uniform evolutionary processes. In this section we first investigate typical Darwinian models and point out their strengths and weaknesses. Later, we discuss the theory based on hypercycles and explain why hypercycles are much more preferable and adequate models for capturing the nature of evolutionary processes.

Darwin’s principle of natural selection is currently widely used in computational models of evo-

---

\(^2\)The global, overall purpose.
olutionary systems for optimisation or simulation purposes. However it is quite a misrepresentation to
treat the principle of natural selection in the category of first principles (or axioms). Natural selection
is rather a consequence of the properties and dynamics of a population of objects subjected to specified external constraints. The main objective of the work of Darwin and Wallace was to provide some basic insights into the process of evolution and the phylogenetic interrelations among species, with the assumptions based on the notion of natural selection.

We note that there are some inherent properties of Darwinian systems that constrain their applicability to open-ended evolutionary systems. Darwinian systems rely on a concept of environment with embedded, self-replicating entities that compete for selection. The Darwinian systems postulates a stable species which compete for selective preferences, and it postulates stable reproduction of the best adopted species. There is a limit to the amount of information content that a stable species can have (Eigen and Schuster, 1979; Wright, 1931). Therefore the evolution of such a system is limited to a certain level of complexity defined by the threshold for maximum information content. Translation of the information inherited by the reproductive material becomes a requirement to allow the evolutionary process to continue. This translation step however is not present at all in Darwinian systems. A detailed discussion on the limit and the concept of quasi-species is presented in (Eigen and Schuster, 1979). We provide a brief discussion on hypercycles in the next section.

7.5 Hypercycles

7.5.1 Definition of the hypercycle

Let us consider a sequence of reactions in which products, with or without the help of additional reactants, undergo further transformations. The reaction cycle or cycle is such a sequence of reactions such that some of the products are identical with the reactant of any previous step of the sequence. The most basic is a three-members cycle, with a substrate, an enzyme, and a product. The enzyme transforms the substrate into an enzyme-substrate, and then enzyme-product complexes, which in turn are transformed into the product and a free enzyme. See Figure 7.1.

In such a case, the cycle as a whole works as a catalyst. Unidirectional cyclic restoration of the intermediates presumes a system far from equilibrium. This can be associated with a dissipation of energy into the environment. Equilibration occurring in a closed system will cause each individual step to be in balance: catalytic action in such a closed system will be microscopically reversible.

Let us now consider a reaction cycle in which at least one of the intermediates themselves is a catalyst (see the work of Kaufmann on autocatalytic nets, e.g. (Kauffman, 1993)). The simplest representative of this category is a single autocatalyst (or a self-replicative unit). A system which connects autocatalytic or self-replicative units through a cyclic linkage is called a hypercycle. Figure 7.5.1 depicts a schematic view of a hypercycle. Compared with a simple autocatalyst or a self-replicative unit (which by its nature represents a flat structure), a hypercycle is self-reproductive to a
Figure 7.1: An example of a three-members catalytic cycle: the free enzyme (E), the enzyme-substrate (ES) and the enzyme-product (EP) complexes all demonstrate a catalytic cyclic restoration of the intermediates in the turnover of the substrate (S) to the product (P).

higher degree. This is because each of the intermediaries can itself be an autocatalytic cycle.

One of the goals of the EVMA is to provide a computational environment where hypercycles can be created, simulated and observed, in computational settings. See Chapter 9 for further discussion on EVM and hypercyclic dependencies.

Figure 7.2: A schematic view of hypercycle. Catalytic cycles $C_i$ form a hypercycle through the consumption and production of free enzymes $E_i$.

7.5.2 Hierarchies

In Darwinian systems all self-replicative units competing for selection are non-coupled. In other words, the selection forces operate purely on a single level. It can be a level of individuals, of species, or of genes. The fact is that the simple Darwinian model operates only on a single level. We have here a conservation of a limited amount of information, which cannot pass a specified threshold (see
the discussion above). Hypercyclic systems also deal with similar evolutionary selective pressures. However in this case we also deal with integrating properties, and this allows for cooperation of otherwise competing units to develop: hypercycles are able of establishing higher-order linkages. When inter-cyclic coupling is established, individual hypercycles may form hierarchies. In other words the basic unit of selection is not a single hypercycle; it is a whole chain of interrelated hypercycles (Eigen and Schuster, 1979).

Similarly, in the EVMA, although lower level selective pressure may lead to survival or extinction of certain individual programs (cells), the basic unit of selection is not a single cell. The successful units will invariably form long interlinked chains. It will be a network of many individual cells contributing to the overall success. Only the entire network can then survive, as a whole.

7.6 Autopoiesis

Gdyby wszystko było tym tylko, czym jest, i gdyby każdy element istnienia nie rwał się gdzie indziej, czyź nie byłoby to równoznaczne z absolutną nicością? Dlatego to całość istnienia w pojęciu witalistycznym trzeba przyjąć nie jako zbiorowisko istnień, a ich organizacje, coś w rodzaju rośliny. Przyjęcie bowiem jednego jedynego istnienia implikuje nicość.

(If everything were only what it seems to be and if every element of existence did not try to move somewhere else, wouldn’t that be equivalent of absolute oblivion? That is why the notion of existence in the vitalistic sense is not simply a collection of all living entities, but rather their organisation, like that of a plant. Postulating a single one and only true existence implies nihilism.)

(Witkiewicz, 1992, p.126)

The work on hypercycles has been developed independently to, but it shares common features with, the theory of autopoiesis. In this section we discuss the theory of autopoiesis and draw parallels with the theory of hypercycles.

The theory of autopoiesis has been developed by two Chilean biologists, Humberto Maturana and Francisco Varela, in the early 1970s. There has been a steady increase in interest and in citations numbers since the inception of the theory. Despite the fact that Varela provided computer simulations shortly after developing the theory, autopoiesis has not been widely used in computer science. At first used primarily in theoretical biology, cognitive sciences and systems theory, it made its way and has been widely used in law, social sciences, sociology and many other disciplines (mostly in humanities). In recent years it became more widely known in computer science and has been applied and used in connection with Artificial Life itself (McMullin, 1997).

In recent years the theory has been re-shaped slightly and used as a basis for discussions on new notions of the concepts of biological information and complexity. The original assumptions on the
physical embodiment, central to autopoiesis, can be then considered to be the shortcomings of the theory.

 [...] the whole bionetwork deals basically with self-production activities. The cell, as Changeux puts it, is but “an enormous factory” devoted to the production of molecular machines. In the 70’s it was argumented under the label of ”AUTOPOIESIS” by Maturana and Varela, unfortunately they did not continue the integration of new biomolecular facts and their synthesis was left suffering the usual ”premature closure” ailment, and somehow pretentious and empty for our times. Thus, to continue with Jerry’s taxonomy, the problem is that the self-production network is a mere label for an enormous landscape of molecular interactions and processes of quite a heterogeneous nature. Let me take home the term ”molecular landscape”. 


It is true that the original closed-form autopoiesis formulation does not incorporate the recent biomolecular discoveries. On the other hand, however, the core of the theory has not been changed, as such, and can still be used as a basis for more accurate formulations that incorporate all the recent discoveries. The EVM architecture is one such attempt. It builds on autopoiesis and expands it by some new mechanisms. The EVM theory tries to incorporate other evolutionary notions, such as hypercycles, symbiogenesis, and Rnomics.

In the following sections we provide the introduction to the general theory of autopoiesis, as formulated originally by Maturana and Varela. This will be used as a basis for further discussions on computational autopoiesis, and on the presentation of our EVM architecture.

7.6.1 Machines

We discuss here the notion of a machine from the autopoietic perspective. Note, that the concept of a machine is different in autopoiesis from the computational perspective. We try to draw some parallels and describe where these two concepts overlap and how these concepts differ.

Machines are units which are made out of components. All components are characterised by certain properties capable of satisfying certain relations that determine within the machine (in the unity) the interactions and transformations of these components. The actual nature of components and their particular properties, other than those participating in the interactions and transformations which constitute the machine, are irrelevant and can be arbitrary. In the context of the EVM machines, all the side-effects that do not contribute to the relations characteristic of the unity are not considered to be important and can be any. That is, several different programs can be considered as the same machine, as long as they satisfy all the necessary relationships. This is inherently simple and intuitive in the context of computation and virtual machines. For example any Java interpreter is considered a Java interpreter, no matter what extra functionality that interpreter may posses, or how internally it
has been implemented. Thus there is a certain level of granularity that is mapped to a given notion of a machine.

The organisation of a machine is defined as all the relations which define a machine as a unity and determine the dynamics of interactions and transformations which it may undergo as such a unity. The organisation of a machine does not specify the properties of the components which realise a concrete machine. The organisation of a machine is independent of the arbitrary properties of its components, and a given machine can be realised in many different manners by many different kinds of components. In other words, the organisation is the functional abstraction over the actual physical (or computational) realisation of a given machine.

The structure of a machine is defined as the actual relations which hold among the components which integrate a concrete machine in a given space. This is the actual realisation, or implementation, of a given machine. This represents the actual program of the machine, with all its properties and functionality.

Note that a given machine (machine with fixed organisation) can be realised by many different structures. For example, an organisation may remain constant by being static, by maintaining its components constant, or by maintaining constant certain relations between components which are otherwise in continuous flow or change.

### 7.6.2 Definitions

An autopoietic machine is defined as a unity by a network of production, transformation, and destruction of components which: (i) through their interactions and transformation continuously regenerate and realise the network of relations that produced them, and (ii) constitute the machine as a concrete unity in the space in which the components exist, by specifying the topological domain of its realisation as such a network (Maturana and Varela, 1980). This somewhat abstract definition, in the context of the EVM architecture, means that an autopoietic machine is such a machine that continuously regenerates and realises all the necessary components of the interlinked network of dependencies, and by doing so, maintains all the necessary dependencies and computations. In other words, it maintains itself, it achieves homeostasis.

A living system is considered to be a unity in physical space. It is an entity topologically and operationally separable from the physical background. It is defined by an organisation that consists of a network of processes of production and transformation of components, molecular and otherwise, that through their interactions: a) recursively generate the same network of processes of production of components that generated them; and b) constitute the system as a physical unity by determining its boundaries in the physical space. As defined above, this organisation is called an autopoietic system.

An autopoietic machine is an homeostatic, or rather a relations-static, system which has its own organisation as the fundamental variable which it maintains constant.

A machine whose organisation is not autopoietic does not produce the components that constitute
it. The product of such a machine is different from the machine itself. The physical unity of such a machine is determined by processes that do not enter into its organisation. Such a machine is called allopoietic (Maturana and Varela, 1980). Allopoietic machines have input and output relations as a characteristic of their organisation: their output is the product of their operation, and their input is what they transform to produce this product. The phenomenology of an allopoietic machine is the phenomenology of its input-output relations. The realisation of allopoietic machines is determined by external processes, and these external processes do not enter into the machine’s organisation.

### 7.6.3 Identity

The important aspect of an autopoietic system, as defined in (Maturana and Varela, 1980), is that it remains invariant in its organisation. The system itself can be deformed by external circumstances, but its internal organisation is supposed to remain invariant. Any change in the autopoietic organisation is equivalent to the loss of identity, and system disintegration - in other words: death.

A living system is defined as a unity by its autopoietic organisation. All the transformations that it may undergo without losing its identity are transformations in which its organisation remains invariant. All autopoietic systems are therefore homeostatic. They maintain their own organisation constant through their operation. All the unitary phenomena of an autopoietic system are subordinated to the maintenance of its autopoiesis.

The concept of identity as proposed by Maturana and Varela does not go well with the concept of hypercycle, or even with the concept of higher autopoiesis (discussed later). The problem is that the flat and homeostatic notion of a single level is not really translatable into a flexible and dynamic notion of continuous change in hypercycle or higher autopoiesis. We are not going to introduce any new notion in place of the above-proposed notion of living system identity; however, we want to stress that this is a purely operational notion that does not really have any impact on the proposed theory of living systems presented here. The stress here is on hypercycles and higher autopoiesis. On this level the traditional concept of identity is not applicable anymore – the identities change depending on the context, and on the level that is used to compare the identity to. Therefore, only in specific, flat structural analyses does the concept of identity make sense and offer a context in which it can be fruitfully used.

### 7.6.4 Hierarchies

The term “hierarchy” has not been used directly within the scope of the theory of autopoiesis. Instead, there is an equivalent term used with the same semantics: higher autopoiesis. The standard interpretation of the theory of autopoiesis differs from other models by the lack of hierarchical structure. According to Maturana and Varela, there is no place for hierarchy, because an autopoietic machine, by definition, has a flat structure and is independent from other machines. Yet, higher autopoiesis,
even within the theory of “flat autopoiesis”, opens the door to unify the higher autocatalytic organisation, hypercycles and autopoiesis.

If the higher order autopoietic system undergoes self-reproduction (through the self-reproduction of one of its component autopoietic unities or otherwise), an evolutionary process begins in which the evolution of the manner of realization of the component autopoietic system is necessarily subordinated to the evolution of the manner of realization of the composite unity. Furthermore, it is to be expected that if the proper contingencies are given, higher order autopoietic unities will be formed through selection. In fact, if coupling arises as a form of satisfying autopoiesis, a second order unity formed from previous autopoietic system will be more stable, the more stable the coupling is. However, the most stable condition for coupling appears if the unity organization is precisely geared to maintain this organization, this is, if the unity becomes autopoietic. There is then an ever present selective pressure for the constitution of higher order autopoietic systems from the coupling of lower order autopoietic unities [...]

(Maturana and Varela, 1980, page 111)

To a certain extent the above description bridges the theory of autopoiesis and the theory of hypercycles. At a certain level (higher level) of organisations, these two theories are trying to capture higher-order self-organisation phenomena. In this context, in the theory of autopoiesis and hypercycles, unlike many other theoretical models of the process of life, the process of evolution is simply a side-effect, a consequence of limited resources, not the prerequisite that drives the process itself. Whenever we deal with restricted resources, we have the selection and evolutionary pressures naturally occurring within our computational models. It is however important to remember that life (or precisely: higher-order autopoietic systems, or hypercycles) would exist even if the process of evolution were not to occur in a system as such. Life, in the context of autopoietic hypercycles, is a certain property, a certain type of organisation. Evolution is the process of continuous change of this living organisation. That change can be driven by limited resources, and this is what evolutionary Darwinism is all about. But this change can be driven by many other phenomena, some inherently present in the dynamics of the organisation itself. This is what the EVM theory is trying to provide as a basis for modelling and investigations. We have explored some of the issues already in Chapter 2 and we will come back to this theme in Chapter 9, where we will discuss some of the experimental results.

### 7.6.5 EVM and the theory of autopoiesis

The name *evolvable* (from the EVM acronym) is perhaps poorly chosen, as we do follow the Maturana’s model (Maturana and Varela, 1980) in the sense that evolution is a side-effect, a consequence,
not the prerequisite of life. However, as we deal always with limited resources and restricted computations, we have the selection and evolutionary processes naturally occurring within our computational models.

Our model departs from Maturana’s work in some important points. First, we do not stress the notion of “physical space” (which, by the way, Maturana does not define formally in his work (Maturana and Varela, 1980)) as a prerequisite of autopoiesis. For the theory to work, it does not actually matter what sort of “space” is being used for synthesis and research of autopoietic processes. It is just like the situation with different branches of science: some are more abstract than others, but the aim is always the same: to make predictions about physical reality. For some fields it makes sense to ground the theory in a real physical reality, for example in case of biology; but for some it does not help at all, for example in mathematics. We believe, that a general abstract theory of life which could be applied easily to both virtual and physical spaces is of great interest to progress the research, not only in theoretical biology, but also in evolutionary biology, computer science and information science. Hence, we do not require the “space” to be of a physical nature.

Second, we do not require the “machines” from Maturana’s original autopoietic theory to be Turing-like. We believe that this is too restrictive, and we allow any computing machine to be used as a component in the autopoietic system.

Third, we use the notion of hypercycles of autopoietic systems. This is the biggest innovation, which unifies these two theories into one. It will be explained in more detail in the following chapters.

7.7 Artificial and Natural life

7.7.1 Causality and relativism of life

When discussing whether a given object (or program) is alive or not, one sometimes investigates the causes of existence of the object. The answer is not always a simple “yes/no”, alive or not alive. The answer depends on the frame of reference. In some models (Orgel, 1973; Chaitin, 1979) the question itself might be irrelevant, without specifying a concrete frame of reference. In our view it is often appropriate to identify an object by using operational notions (both, for object identity and object boundary). The notion of an object (or program) itself is an approximation, an operational tool used to describe certain properties on certain levels of operation. Overall, everything can be interlinked with everything else, and the whole living universe (or the EVM Universe, in the context of the EVM architecture) is just one single giant living structure, higher-order autopoiesis, or hypercyclic autopoiesis. It should be remembered that the traditional notions of cells, organs, organisms, species, etc., are just mere approximations that allow us to discuss and manipulate the overwhelming complexity of this single interconnected blob.

The main, and in our opinion the only, difference between natural and artificial processes of life is their respective origin. The main characteristics of the process of natural life are derived from
expectation of the process to be spontaneous. The process of natural life needs to be an inherent part within a given system, and to be inherently driven by the physical laws of the system. The only path available for us to replicate such characteristics is to build an artificial environment in which such spontaneous and self-sustained processes occur. Note here, that in such a case, the obtained processes would be expected to exhibit properties characteristic of the natural phenomenon of life.

This is one of the main objectives of the EVM Universe. The majority of artificial life systems, however, use a different approach. The human designer sets up the experimental environment such that it leads to a particular, expected behaviour. This might be called a “typical” artificial life system. In such a case the main characteristics of (typical) artificial life processes are derived from the actual designed setup and may differ from the natural life characteristics. This is an important distinction, and this is what researchers of artificial life should be aware of.

Based on the above definitions, we obtained the following simple taxonomy (see Table 7.7.1):

- **Type A**: natural environment; natural process of life. This is what we refer to, in short, as the *natural process of life*. This is where the process of life is truly spontaneous and self-sustained, placed in a natural (physical) environment.

- **Type D**: artificial environment; artificial process of life. This is what we refer to, in short, as (typical) *artificial process of life*. This is what most of the current artificial life systems are trying to achieve and are modelling processes which exhibit some of the properties of the natural life.

- **Type B**: artificial environment; natural process of life. This is what we believe researchers of the field of artificial life should aim at. This is when we try to replicate truly natural and spontaneous processes of sustainable self-organisation, adaptation, and life. We refer to this type as *proper artificial life*.

- **And, Type C**: natural environment, artificial process of life. This is where research is leading us too, i.e. developing semi-alife, autonomous systems embedded in natural physical environments, personal assistants, robots, wearable computing devices, and so on.

<table>
<thead>
<tr>
<th></th>
<th>natural environment</th>
<th>artificial environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>natural life</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>artificial life</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

In the context of the EVM Universe, we focus on type B, although certain outputs of this research can be used within type A and D, as well. For example, our currently investigated learning strategies can be used for systems type D, and the architectural design, the environment model, and the related algorithms can be used within type B artificial life research.
7.7.2 Spontaneous autopoietic hypercycle

There remain some fundamental questions, e.g.:

- Is it possible for a higher level autopoietic entity to appear spontaneously?
- What environmental conditions and circumstances need to be present for the spontaneous appearance of a higher level autopoietic organisation?
- Can we obtain higher level autopoiesis in artificial Turing computational environments?

It appears that above a certain organisation and complexity threshold, it is plausible for an apparently spontaneous generation of autopoietic entities, within Turing-level computational universes (see (McMullin, 1997). However, spontaneous generation of higher-level autopoietic organisation is a completely new level of complexity and there is no work to date available that would address that issue.

Typically, we deal with entities that are created from outside of the system: in terms of Maturana and Varela language, we deal with allopoietic systems. For instance this is how humans gain knowledge and experience and how artificial evolutionary systems are designed. However, in artificial systems of type $B$, we are interested in the spontaneous generation of information-collecting entities, which are capable of fundamentally different kind of organisation: hypercyclic autopoiesis. This is a complex issue. Although many of the elements of the proposed EVMA are geared toward investigation of the spontaneous hypercyclic autopoietic systems, that issue itself has not yet been directly addressed in the current state of the work. We hope that the current state of the theory and the architecture can stimulate interest and research in this area.

Currently, the only autopoietic behaviour that can be viewed as hypercyclic within the EVMA has used the external environment and it has been hand coded. Due to the limited computational capabilities of the execution environment, we were unable to identify a minimal autopoietic hypercyclic model within the Turing-machine-like environment equivalent to the cellular-based autopoiesis discussed by (McMullin, 1997). We believe, however, that our approach is a promising path to pursue and further refinement of the concrete EVM Universe implementation should lead to progress in modelling and investigating computational hypercycles and autopoiesis. Further investigations in the area of hypercyclic behaviour is planned as future work.
Chapter 8

Learning how to learn
8.1 Biological learning

8.1.1 Early theories of learning

Most of the modern theory of learning originated in the work of Watson (Watson, 1907) and then Pavlov (Pavlov, 1927). The main ideas here are the notion of conditioning, and *conditioned stimulus* with a *conditioned response*. In this model, a reward system is a fundamental element, and it is present in any of the early work on conditioning. The basic idea is that positive feedback is an essential part of conditioning, i.e. given a conditioned stimulus positive feedback is employed to achieve the conditioned response. In other words, when a positive reward accompanies a particular response to a stimulus over many repetitions, then that particular response will become conditional to the stimulus. In experimental studies where conditioning trials were not accompanied with positive feedback (rewards) the expected conditioning was not achieved. The early work on learning and conditioning perceived (or projected) the living organisms as simply structured mechanistic systems. Contemporary biology and psychology departs from this simplistic model and treats associative processes as being fundamental to all learning. In contemporary studies on conditioning, the exclusively mechanistic view is being redefined to account for richer and more complex dependencies. These dependencies between the stimulus, conditioning and positive and negative feedback loops can form a highly complex, interlinked, and hierarchical network. It is important to note though, that the role of a (positive and negative) feedback continues to play the central role in contemporary theories of learning as it did at the beginning of 20th century.

The reward system in biological studies is almost always based on exclusively binary (all or nothing) signals. A subject receives the full reward if the expected conditioned response is achieved, or otherwise, there is no reward at all. There is no concept of partial reward for partial conditioned responses (e.g. an animal reaches a food source in a maze, or it does not). The process of learning is equivalent to the process of associating the conditioned stimuli with conditioned response. It is important to distinguish this type of learning, from other machine learning techniques where the feedback mechanism is provided in a form of a gradient, i.e. there are partial rewards for partial responses. We will discuss these issues in more detail later in this chapter.
8.1.2 Law of Effect

The Law of Effect (Thorndike, 1911) tries to capture the idea that those aspects of behaviour which satisfy certain needs of an organism will tend to be repeated, whereas those which do not satisfy any needs will tend to be omitted from future behaviour. Intuitively, it is clear that something like the Law of Effect must operate on different levels of biological organisations to facilitate the process of learning and adaptation. There is evidence that reinforcement, both, positive through rewards and negatively through punishments is necessary for the process of learning to occur. The question is which features of the environment function as positive and which as negative reinforcers. The difficulty of setting up classes of reinforcers is mainly that a reinforcement is dependent on the context in which it operates. The reinforcement in one context will not be the same in another. In fact, positive reinforcement in one context may have exactly opposite effect in another context. For example, when hungry, the smell of food provides positive reinforcement and increases subject’s desire to eat. However when a subject is full and satisfied, the smell of food may have quite the opposite effect. Detailed discussion on biological aspects of reinforcement in the context of cybernetics is provided, for example, by George (George, 1965).

Not only do the reinforcers work depending on the physical context. They also operate over time delays. Often, a reinforcement (in the form of a reward or punishment) does not occur immediately after a cause. Both, negative feedback and the positive reinforcement can take substantial amounts of time to occur. The positive (or negative) feedback, may be immediately proceeded by events that have no causal relationship to the reinforcement. For example, a dog might have chewed the owner’s shoes after which he went in front of the house and lay down for some time. Once the owner notices the damage the dog makes, and punishes the dog, it may not be apparent for the dog (properly associated) what the reinforcement is actually for (chewing on the shoes or lying in front of the house). Biological organisms are surprisingly well-attuned in selecting the appropriate causal relationships and filtering out the events that do not have causal relationship to the rewards (punishments). They usually infer (most of the time without any cognitive or conscious reasoning as such) correct context from cues and regularities of the environment and the reinforcement itself.

8.1.3 Short term vs. long term reinforcers

Sometimes, short-term positive (or negative) reinforcement is cancelled out by a long term negative (or positive) reinforcement. Biological organisms seem to have little trouble learning the best strategy that maximises the overall positive feedback, but there are certain exceptions. Sometimes, organisms seem to be wired in such a way that short-term positive reinforcement is so strong that it overrides any long-term negative reinforcement that the organism may obtain. In extreme cases, such a short-term override may eventually lead to the death of a biological organism. One such example is studied through the intracranial self-stimulation (ICS) experiments (Bozarth, Gerber, and Wise, 1980). In a typical ICS experiment subjects (e.g. rats or monkeys) are taught to activate self-stimulation (for ex-
ample dopamine or cocaine stimulation) directly to their brains through manipulation of certain physical objects (such as a switch or a button). After initial training, subjects are incapable of breaking the cycle and continue to perpetually activate the self-pleasuring stimuli. The short-term reinforcement is so strong, that the subjects can ultimately pleasure themselves to death (Bozarth et al., 1980).

In normal (or a majority of) circumstances, however, the short and long term reinforcers work together and complement each other. Organisms’ adaptation can be viewed as a balancing act between different, sometimes conflicting, reinforcers.

### 8.1.4 Multi-task learning in biology

Initial studies on conditioning were conducted in isolated laboratory settings, where single stimuli and single reward mechanisms were used. For example early Pavlov work on conditioning food responses in dogs through a particular sound, or light pattern fit into this category (Pavlov, 1927).

Despite this early experimental work, it is clear that biological organisms in their natural environment are not presented with a single stimulus and a single reward mechanism that enforces a single conditioned response. Hence a simplistic model of conditioned stimuli and reinforced responses is useful as a first approximation, but it does not capture the entire spectrum of all possible dependencies and learning mechanisms. Simple models seem not to be sufficient for modelling learning in biological organisms and more elaborate models of learning are now being proposed (see e.g. (Albus, 1975)).

Biological organisms (and human beings in particular), are continuously confronted with a complex web of interconnected tasks. Some of the tasks complement each other, some contradict, some can be done, one after the other, in any order, some must be conducted concurrently, and some must be conducted sequentially in some order. In real environments many independent tasks must be completed (or carried out) concurrently, and it is not easy to filter and identify a complex delayed reward (with negative and positive feedback) through such a complex interaction web with the environment. Some psychological studies suggest that humans are particularly effective in generalising, i.e. discovering patterns and rules, in such complex and dynamic environments. It has been shown that humans can often correctly infer a general rule even from a single training example (Ahn and Brewer, 1993; Moses, Ullman, , and Edelman, 1993). Possibly the main reason behind such excellent generalisation skills is the ability to store, analyse and exploit regularities from the enormous amount of past training data accumulated throughout an entire lifetime (of a given individual, or entire species). In living systems, in general, individuals (populations, species) face a continuous stream of problems to solve with respect to many different levels of organisation. To solve a new problem, the ability to reuse all the previously obtained experiences is essential. Some of these experiences are learned throughout a lifetime of a given individual. Some however, are “hardwired” and become instinctive through the evolution of a species itself.
8.1.5 The pieces of the puzzle

Based on the above discussion, a complex picture of biological learning systems emerges. We will highlight below some of the main points.

- Reinforcement is the key to learning.

- Reinforcement works often as a binary system; a given reinforcement is present, or not. In biological systems there is often no gradual or partial negative or positive reinforcement. The reinforcement is simply on or off (the dog was either punished for chewing on owners shoes, or was not punished).

- Reinforcement is often not immediate; it frequently follows the triggering organism actions with a delay.

- Reinforcement is context-dependant; the same stimuli can provide opposite or contrasting reinforcement in different contexts.

- The same stimuli can have opposite short-term and long-term effects, hence short-term negative reinforcement can lead to a long-term positive feedback, and vice versa\(^1\).

- Generalisation skills and life-long learning appears to be essential in coping with the complexities of the reinforcement biological learning.

There are many unknown issues and open questions in the area of learning. The emerging theories and overall picture of biological learning that we have explored so far are complex, highly dynamic and appear to have many interlinked dependencies. We perceive the learning process occurring simultaneously at different levels of biological organisation, e.g. both on the base level (to learn the immediate tasks at hand) and at the meta-level – to learn the abstractions, generalisations and biases that can be transferred, directly or indirectly, to enhance one’s abilities to learn new tasks. In general, learning how to learn in biological systems is essential. This leads to a meta-meta-...-meta-learning chain. These and related notions and concepts lie at the heart of the EVM architecture. Binary reinforcement, life-long learning, and meta-learning – these are the three fundamental properties that our EVM model tries to investigate and exploit through:

- a hierarchically organised computing architecture;

- reflection and reification mechanisms allowing the modification of one’s own internal processing (learning);\(^2\)

\(^1\)For example, for endurance and physical training in general, subjects will feel pain and the immediate reinforcement is negative. However, in a longer perspective, serotonin and other chemicals are released in the brain that generally makes subjects positive and helps generate a good mood (even though the physical pain persists in the muscles). The long term benefits of physical training are the third, long term positive reinforcers, that make the subjects perpetuate the exercise, and continue to overcome the initial negative reinforcement.
• keeping the history of earlier discoveries (memory);
• reusing learned skills in different contexts (exaptation);
• the ability to deal with delayed rewards/punishments;
• the ability to generalise previous experiences;
• meta-learning constructs built-in into the architecture (see Chapter 3).

8.2 Machine learning

Generally, the term machine learning refers to various aspects of computational mechanisms for learning. There is a large body of research work concerned with the optimisation, prediction and the use of various heuristics to solve problems by means of algorithmic computations. In our work, we are concerned exclusively with a small subset of machine learning inspired by highly dynamic biological learning systems that are characterised by life-long learning process. These reinforcement learning systems form a hierarchical short and long-term reward/punishment structure. We will briefly review the state of the art through discussion and analysis of various existing techniques. Then we provide a discussion of how the EVMA and EVMI address the issue of life-long reinforcement. In the next section, we provide a discussion and experimental results of the application of the EVMI to some of the typical problems tackled in the field.

8.2.1 Early reinforcement learning

Reinforcement learning concerns agents that sense and act in their environments, trying to learn to choose optimal actions to achieve a maximisation of reward intake. Each time the agent performs an action, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. The task of the agent is to learn from the possibly delayed rewards to choose sequences of actions that produce the greatest cumulative reward.

The study of reinforcement machine learning is as old as the theory of algorithmic computation, itself. In his article Turing (1948) studied a special type of unorganised machines (u-machines) that to a certain extent can be seen as precursors of contemporary evolutionary and computational learning systems. He advised three types of unorganised machines: A-type, B-type and P-type. These machines consist of randomly connected two-state machines whose operation is synchronised by means of a central digital clock. A-type machines are described in detail in Section 4.3. By the application of “appropriate interference, mimicking education” a B-type machine can be trained to “do any required job, given sufficient time and provided the number of units is sufficient” (Turing, 1948). His P-type unorganised machines, have “only two interfering inputs, one for ‘pleasure’ or ‘reward’ […] and the other for ‘pain’ or ‘punishment’ “. Turing studied P-types in the hope of discovering training
procedures analogous to the kind of process by which a child would really be taught. Later, Turing said: I have done some […] experiments with one such child-machine, and succeeded in teaching it a few things. A-type, B-type and P-type machines are described more fully in (Turing, 1948).

The foundations for many machine learning techniques have been also laid by the work of John Holland, (Holland, 1968, 1975). Holland is considered a precursor and father of many machine learning techniques, including reinforcement learning and evolutionary computation. Formally, in the general case, we can model the state of the environment (and agent) in time $t$ as $s_t \in S$, where $S$ is the entire possible state space. An agent performs actions $a_t \in A$ that generate the next state $s_{t+1}$ through the state transition function $\delta(s_t, a_t)$. All the future states $s_{t+2}, s_{t+3}, \ldots$ can be influenced by a given action $a_t$. The reward function of the state of the environment, in the general case, $r(s_t, a_t) \in R$ can vary in time, too. The task of the agent is to choose such a sequence of actions $a_t$ that will maximise its reward intake over time. This can be expressed by different objective functions. The two most common are:

1. maximisation of discounted cumulative reward, $\max \sum_{i=0}^{\infty} \gamma^i r_{t+i}$, where $\gamma$ is a parameter from the range $[0, 1]$ representing the memory of the previously collected rewards – the larger the $\gamma$, the longer the system remembers previously achieved rewards. For small $\gamma$ the older rewards influence the cumulative score less and less.

2. maximisation of average reward, $\max \lim_{h \to \infty} \frac{1}{n} \sum_{i=0}^{h} r_{t+1}$.

More details related to reinforcement learning can be found in the literature, for example (Mitchell et al., 1984), pages 367–387.

8.2.2 Life-long reinforcement learning

The reinforcement learning models are often applied to autonomous intelligent agents (e.g. robots) which try to operate in complex dynamical environments. Such situations require that an agent learns several related tasks within the same environment, using the same sensors and actuators. There is no possibility of re-setting the agent state – the agent must continuously sense, learn and act. The agent may in fact never revisit any of its previous states twice in its entire lifetime. This is different from some other machine learning methods, such as, for example, genetic algorithms, which generate a hypothesis to be tested on a single isolated and idealised testcase. After that, a new hypothesis is tested on exactly the same isolated test case, again, and again. In life-long learning this is not possible – any given (new) hypothesis cannot be tested on any of the old isolated and idealised test cases, because there is no possibility of reverting the environment precisely to any of its previous states. One may say that in life-long reinforcement learning, the time flows in one direction. There is no possibility of resetting the clock. Mobile robots cannot test a hypothesis and move from a given location only to restart again with a different hypothesis from the same location again. Once a
decision has been taken, the situation has changed, and new conditions need to be taken into account, a new environment state must be sensed, and new decision needs to be taken.

8.2.3 Multi-task systems

We will use the terms learning tasks or solving tasks/problems interchangeably. Tasks that need to be solved can be provided to the system sequentially (Schmidhuber, 1999, 2004), or the tasks can be provided all at once, in a group (see Chapter 9). When the tasks are provided in sequence, we talk of incremental learning. Incremental learning usually takes advantage of the fact that earlier tasks (typically easier) can provide beneficial biases towards solving later (usually harder) tasks. It is common practice in incremental learning to use binary reinforcement mechanisms (as discussed earlier in Section 8.1).

Multi-task learning is an area of machine learning which studies methods that can take advantage of previously learned knowledge by generalising and reusing it in the course of solving a set of possibly related tasks. This is closely related to the notion of incremental learning. Incremental learning is a special case of multi-task learning, where incrementally more difficult tasks are provided to the system sequentially, in such a way that the system can tackle easier tasks first and gradually solve more and more difficult tasks as time goes on. Multi-task learning is a more general term, where the difficulty of individual tasks may or may not be of concern. There is research work conducted within the area of multi-task learning applied to various domains and experimental settings and employing different learning mechanisms, eg. Xue, Liao, Carin, and Krishnapuram (2007). If the tasks share some similar internal structure, the learner may exploit these regularities and find it easier to learn them together rather than in isolation (Thrun, 1996; Thrun and Pratt, 1998). Caruana and Baxter demonstrated that learning multiple tasks within an environment of related tasks can potentially give much better generalisation than learning a single task alone (Caruana, 1997; Baxter, 2000).

8.2.4 Meta-learning

Meta-learning (Giraud-Carrier, Vilalta, and Brazdil, 2004) is a relatively new area of research within the machine learning and data mining fields. Researchers within the field try to understand the process of exploiting knowledge about learning and build algorithms that can improve their own performance of the learning process itself. Recent advances provide the foundations for constructing meta-learning assistants and task-adaptive learners. One of the main motivations is the fact that successful applications of complex software systems in the real-world require a continuous adaptation to new needs, changing requirements and changing contexts. When a given model fails to perform efficiently at a given time, one would expect the learning mechanism itself to re-learn, taking into account previously learned experiences (Giraud-Carrier et al., 2004). Meta-learning capabilities are necessary in exploiting cumulative expertise gained from previous experiences. This is particularly evident in biological learning and in artificial multi-task learning systems.
One of the contributions that inspired our EVM architecture is the work related to the Optimal Ordered Problem Solver, OOPS (Schmidhuber, 2004). OOPS is based on the sequential exploration of program search spaces associated with Levin’s universal search algorithm (Levin, 1973). OOPS uses self-delimiting binary programs\(^2\) and explores the space of programs by trying to find one that provides a solution to a given target set of problems. Each new problem is provided to the system only after the previously provided problem has been successfully solved. Each solution found (to any of the previously solved tasks) is stored in the system’s storage. Schmidhuber work extends earlier work on bias-optimal search algorithms (Hutter, 2002) (for bias-learning see also (Mitchell et al., 1984)). Inherently, such search must deal with the trade-off between:

- **exploration**, that is the search for completely new programs; the search starts from an empty program, and explores all the possible programs from scratch;

- **exploitation**, search for variant solutions; the search tries to reuse already explored subspaces of programs, and use some (or all) of the already gained experiences.

The assumption here is that exploiting experiences collected in previous searches can solve the target problem faster. This is because the learning process can exploit any regularities that have been discovered earlier. In the context of OOPS, earlier discovered programs are stored and they provide (partial) solutions to other problems. In OOPS, meta-knowledge is stored in the form of candidate program solutions. The system uses this information and tries to exploit it for incremental self-improvement. For example, the task of performing long addition and multiplication\(^3\) can be split into individual tasks concerned with additions, and tasks concerned with multiplications. Programs capable of addition and programs capable of multiplications are then stored in the OOPS framework and are used as first-order primitives when searching for solutions to consecutive problems. An important difference between EVM and OOPS is the fact that newly discovered programs in OOPS are simply added into the primitive instruction set. This, for a large number of sub-tasks typical in long life-long reinforcement learning will lead to an exponential growth of the program search space, due to the increase in the instruction set. In the EVM this has been addressed through the notion of

---

\(^2\)Such programs are binary strings with a specific length encoded into the program itself, or with special end-of-program markers, that identify uniquely an end of a given program in a binary sequence. Note that a given program may end anywhere in, potentially, an infinite string of bits. This is in contrast with ordinary binary programs, which start at the beginning of a bit string and end at the end of the bit string. The advantage of self-delimiting programs is that they can be spliced together and will not change their semantics.

\(^3\)Long arithmetic operation means an operation that spans numbers longer than what the single arithmetic unit can handle in a single operation. Consider an arithmetic unit that can handle a 1-digit binary operation of subtraction and addition with a single extra carry bit. If one now tries to add 8-bit long words, the operation would be considered long addition, and the arithmetic operation would need to be chained and performed multiple times on the simple 1-digit arithmetic unit. It is similar to adding for example two 25-digit long decimal numbers with a piece of paper. One needs to understand the addition up to number 19 (adding two arbitrary decimal digits) to be able to perform long addition on arbitrary long decimal numbers, with a pencil and piece of paper.
computational levels. Newly discovered instructions can be added to the primitive level, just as in OOPS. But, they can also be added to new levels created and managed by the EVMK itself. Unlike OOPS, the meta-management of instructions allows the EVM to manage the complexity and trim the program search space.

In the context of meta-learning and to address the problem of exploration vs. exploitation, one possible approach is to split the search process into two, as in (Schmidhuber, 2002b). One part of the search process tries to exploit already built (partial) solutions, and performs traditional exploration/exploitation of the search space. At the same time, the other part of the search process, concurrently, explores the program generators space (explores the meta-search). For simple tasks the former will be more efficient and will find a solution faster. For complex tasks, the time wasted on traditional search (not the meta-search) is insignificantly small compared to the time that meta-search techniques take; therefore no substantial inefficiency can be expected. Other improvements on this scheme may be possible in a specific subclass of computational tasks. Such improvements must be incorporated into the search process in an automated and adaptable way in order to exploit the underlying regularities of the problem, and the search process itself.

8.2.5 Applications and advantages

From a practical perspective there are many problem domains that can be viewed as sets of related tasks. For example, speech recognition may be decomposed along many different axes: words, speakers, accents, etc. Face recognition represents a potentially infinite domain of related tasks. Medical diagnosis and prognosis problems using the same pathology tests are yet another example, as well as time-series prediction, database mining, autonomous service robots, personal software assistants, etc (Baxter, 2000). See also (Caruana, 1997) for more considerations about applications. Multi-task learning techniques have shown promising results when applied to artificial neural networks (Caruana, 1997; Thrun, 1996). In these models neural nets that learn different tasks usually share a common hidden layer that successfully exploits regularities and common properties across multiple tasks.

In most of the existing artificial learning systems, however, multiple-task problems are usually translated into single-task problems. A potentially rich source of information about the problem is then omitted and the possible area for exploitation, both, for multi-task learning and meta-learning is lost. One of the examples of translation of multiple tasks into a single composite task can be observed in the area of evolutionary computation (EC). In EC a set of objectives from a single problem is handled through multi-objective fitness evaluation mechanisms. Despite the fact that the fitness landscape may have multiple extrema, the methods are usually not applicable to more than a single problem at hand. Generalisation and reuse of partial knowledge discovered and knowledge discovery (on the meta-level) are not exploited.

The main advantage of multi-task and meta-learning systems is the ability to reuse previously acquired knowledge and to exploit any regularities across different tasks that can be generalised.
Solutions, or partial solutions of previously solved problems, can be valuable in solving new, unsolved yet (and often more difficult) tasks. The system is designed to shift its bias accordingly to search for a hypothesis space that contains good solutions to many of the problems in the environment. By transferring knowledge across related learning tasks, a learner becomes more experienced and generalises better. We have observed that through our experimental studies that will be discussed in more detail in Chapter 9.

### 8.3 Self-learning system

All our experiences in AI research have led us to believe that for automatic programming, the answer lies in knowledge, in adding a collection of expert rules which will guide code synthesis and transformation.

Lenat (1984)

Although Lenat’s formulation relates to human-experts formulating meta-rules and search bias for a machine-driven search mechanism and code synthesis, it is also applicable to fully automated learning systems. The biases, the rules that can guide code synthesis, are essential in any non-trivial computational search process. Our aim is to develop an adaptable, self-regulating system, in which the expert rules or search bias are both acquired on-the-fly, during runtime, based on the binary feedback the system obtains from the outside environment. In other words, the biases and the expert rules are obtained through a series of test cases.

It is our aim to provide an architecture that facilitates investigations into different possible mechanisms of code synthesis. In the EVMA, we stress the use of symbiosis and exaptation augmented with mechanisms of bias-optimal search techniques. The design and implementation of the basic EVMI is the first necessary step towards the ultimate goal of a truly self-learning artificial (computational) system. By truly self-learning system, we mean an abstracted learning system inspired by biological theory of reinforcement and conditioning. In particular, we stress that the feedback (positive and negative) is (a) provided in binary form only, (b) it is delayed, and (c) context sensitive, and the system starts its learning process without any inductive bias. The system should be capable of building the bias based on the learning process, itself. Note that this formulation of self-learning system departs in many aspects from the traditional machine learning systems, where the inductive bias is introduced ab initio through various elaborate mechanisms. We stress here the abilities of the system to change any inductive bias of the system, through experiences and the learning process, itself.

To express the notion of self-learning more clearly, consider the following abstract architecture, consisting of four elements:

- code generator \((C)\)– this is an active unit of computation that takes a feedback stream as an input and generates the program stream on its output;
• program stream (\(P\)) – this is a sequence of programs that is being tested by an environment; the environment then generates the feedback stream;

• feedback stream (\(F\)) – this is an outcome of evaluating programs in a given environment;

• environment (\(E\)) – this an active unit that takes programs as its input, and generates feedback stream as its output.

In a truly self-learning system, the last component, the environment \(E\), is (by definition) unknown \textit{a priori}. \(E\) may be computable, in which case the sole purpose of the learning system \(C\) is to learn (discover) the functional mapping of the environment, to discover the function \(E\) – i.e. the functional mapping between the programs fed on environment input, and the feedback read on its output. Even if \(E\) is ultimately computable, it may be impractical to discover the actual computation of \(E\) due to computational limitations of the learning computation itself (limitation of \(C\)). In that case, the learning system might try to approximate true \(E\) by \(E_{\text{approx}}\). Note, on more philosophical grounds, \(E\) may be ultimately uncomputable (or unknowable, or undefinable), in which case the purpose of the learning system \(C\) is to approximate the unknown/uncomputable environmental \(E\) to the desired level of accuracy, using a given model of computation \(C\) that the learning system is using.

In soft computing and machine learning systems it is often the case that both, \(C\) and \(E_0\) (initial environment) are given in the form of computable functions. The task of the learning system is then not to discover an unknown environmental function \(E\), but to convert a given form of the function \(E_0\) into a different form, e.g. \(E_1\). In such a case, a purpose of such (somewhat incorrectly called “learning”) system is not to learn \(E\), but to (simply) translate a given computation \(E_0\) into a different computation \(E_1\).

Based on this formulation, most of the mainstream evolutionary computation methods (discussed in detail in Chapter 6) are not truly self-learning. For example, genetic algorithms (Vose, 1999) (and related models, such as original Koza’s tree-based genetic programming (Koza, 1992)) can be seen as an automated computational method of re-coding one solution (\(E_0\)) to a problem provided by an expert, into another solution (\(E_1\)), discovered by the search mechanism. The solution provided by an expert (\(E_0\)) is in a form of phenotype/genotype encodings and fitness function evaluation mechanisms. The search process \(C\) is, in most cases, (extremely) \textit{encoding-sensitive}. Some expert formulations of a problem (a particular solution encoding and fitness function) may lead to fast, effective and efficient re-coding by GAs for a specific problem (\(E_0 \rightarrow E_1\)). Some, on the other hand, may fail to provide a re-coding at all (\(E_0 \rightarrow \text{unknown}\)).

Note, that, although the EVMA tries to situate itself into the category of truly self-learning systems, for any practical reasons it would be infeasible (and inappropriate) to start each time with an

\footnote{There are many different reasons why such a transformation is useful: consider for example two computations computing the number Pi up to 2 million decimal places. One computation is expressed by a short program, but takes many iterations to calculate the expansion. The other computation is expressed as a long (over 2 million bytes) lookup that takes just one instruction cycle to execute.}
empty system without any inductive bias whatsoever. Rather, in practical applications, the EVM system is seeded with previously solved solutions and does in fact re-uses previously gathered learning biases, based on a choice of particular $C$. Unlike GA, where fitness function, crossover, mating etc., are all shared among all different search domains, and the expert bias is encoded almost entirely all within the $E$, in the EVMA the expert bias is entirely encoded in $C$ itself, and $E$ remains to be discovered. The crucial difference between truly self-learning and typical machine learning systems lies in the adaptive ability to modify pre-existing search biases of $C$ during the evolution of the learning process itself ($E$ discovery process).

One example of a set of truly self-learning algorithms that has inspired some of the characteristics of the EVMI is the environment-independent reinforcement acceleration (EIRA) approach proposed by Schmidhuber. We will discuss this example of self-learning in the next section.

8.4 Environment-Independent Reinforcement Acceleration – EIRA

Environment-Independent Reinforcement Acceleration (EIRA) is a reinforcement learning mechanism proposed by Schmidhuber (Schmidhuber, 1994, 1995). EIRA is an environment-independent, single-life (life-long) reinforcement learning mechanism that guarantees life-long performance improvement. EIRA is an example of a truly self-learning system: there can be no initial human-expert bias expressed in encoding the solutions or in shaping the fitness landscape. Theoretically, the system will learn all necessary biases exclusively from the binary feedback mechanism. The executing program monitors its own performance, and it continuously tries different learning policies. Some of the policies that perform worse than any other are discarded, and the system keeps only the useful policies: those that bring more reward per time unit than all the previously tested policies. EIRA provides an elegant mechanism that is embedded in the program language itself.

EIRA inspired and influenced our EVMA, especially the design of the EVM assembly language. One of the objectives in designing our EVM assembly language was to provide easy mechanisms for plug-in reinforcement learning techniques for long-life continuous adaptation and performance improvement. The built-in probability distribution manipulation instructions (see Section 3.6) are in some aspects based on EIRA principles. The EVMA can be seen as a super set of EIRA conceptual learning mechanism. It is important to remember that EVMA extends the learning mechanisms by meta-learning strategies that can work concurrently within the same EVMA grid together with EIRA. EVMA also provides specialised mechanisms for dynamic updating (forgetting) previously learned or established patterns and program structures.

8.4.1 Reinforcement Acceleration Criterion (RAC)

In EIRA, a system policy is an arbitrary modifiable algorithm that maps environmental inputs and internal states to outputs and new internal states. In dynamic and unknown environments in the real
world, each policy modification process (PMP) occurring during system lifetime may have unpredictable influence on environmental states, rewards, and PMPs at later times. Note that this relates to the notion of multiple interactive machines, discussed in section 4.7.4. Although Schmidhuber, the author of EIRA did not pursue this line of research, we have identified that EIRA, MIMs and EVM share common properties that distinguish them from normal Turing-level computing (for details see Chapter 4).

Schmidhuber proposed the reinforcement acceleration criterion (RAC) as a way of measuring performance improvements. At a given time, the RAC is satisfied if the beginning of each completed PMP that computed a currently valid policy modification has been followed by long-term acceleration of average reward intake. In other words, the rewards rate of a new policy must be higher than those of the previous policies. The system keeps only those probability modifications computed by useful self-modification programs: those which bring about more payoff per time unit than all the previous self-modification programs.

Figure 8.1: A meta-instruction modifies the system policy. It launches a new PMP. A checkpoint-instruction triggers a restoration of all the previous policies until one is found that satisfies RAC.

At special instants, called checkpoints, EIRA will restore all the previous PMPs until it finds one that fulfills RAC (Figure 8.1).

According to Schmidhuber (Schmidhuber, 1995), there are many ways of implementing EIRA. Here we describe one possibility to apply it to cell specialisation in the EVM system (for details on EVMI see Chapter 3).

Let every EVM cell contain one program $P$, that is executed in a continuous fashion (in a never-ending loop). $P$ can call up its neighbours and during its own program execution, execute the neighbour programs too (finite number of iterations of the loop, in most cases just one iteration). $P$ has the ability to modify itself, through self-referencing. Because $P$ has the ability to modify the way it modifies itself (meta-learning), there is no intrinsic need for any other learning mechanism to be added on top, other than to provide convenient expert knowledge or pre-design search bias. There is no need for traditional EC techniques, either – no need for special operators, such as mutation or crossover (GAs). The system is able to shift its inductive bias in a universal way. In other words, there is no explicit difference between learning, meta-learning, and other kinds of information processing. As a consequence, the system can reproduce all traditional EC mechanisms as special cases (extensions).

To achieve such a broad unification within the learning process, $P$ needs to have special meta-
instructions that modify \( P \). Executing a meta-instruction corresponds to a PMP, as discussed above.

Following EIRA, one still needs to decide when to perform checkpoints. It can be done periodically by the environment itself (for instance every 100 time steps, if the expected solution to the hardest task is less than 100 time steps). It is more interesting, however, to introduce a special instruction, called \textit{checkpoint-instruction}, that triggers a checkpoint. As \( P \) can modify itself, the frequency of checkpoints is also dynamically and automatically tuned. In theory, the checkpoint frequency may well reflect the difficulty of the tasks. This mechanism will reflect also the dynamism and change-ability of the environment itself.

A policy’s usefulness is defined as the rewards/time ratio (or rewards rate) from the beginning of the policy. The system has to maintain a stack of PMPs. At checkpoints, it keeps only useful PMPs. A PMP is considered useful if it fulfills the RAC, i.e.:

- if there is no previous policy and the rewards rate of its policy is greater than the global rewards rate from the beginning.
- if the policy introduced by this PMP has a higher rewards rate than all the previous policies.

All PMPs are thus restored until one is found that satisfies RAC. For more detail about the implementation of that stack and efficient memory management, please refer to Schmidhuber (1995).

\subsection*{8.4.2 Some comments about EIRA}

EIRA is an environment-independent, single-life reinforcement learning mechanism that guarantees a life-long increase of rewards rate. Moreover, due to its meta-learning facilities, EIRA is universal and requires no initial bias\(^5\).

EIRA is especially well-suited for reinforcement learning problems in which it is impossible to reset the environment to its initial state. This is usually referred to as life-long reinforcement learning; in other words, the time flows in one direction, and the learning environment cannot be simply \textit{reset} to any initial (or previous) state. A typical example is the control of a mobile robot when exactly repeatable training iterations are impossible. Some of our experiments were in such a life-long reinforcement setups, for example the maze experiments (for details see Chapter 9). Usually, with such problems, traditional evolutionary methods are not applicable. Note, random search methods are not applicable, either, because they are incapable of accumulating inductive bias during the search.

There are inherent benefits of life-long reinforcement learning models. They mimic closely biological learning and provide an attractive theoretical and practical framework for experimentation.

EIRA is designed to make optimal use of its computational time/space resources, by exploiting \textit{arbitrary} task-specific regularities (if there are any) (Schmidhuber, 1994). These are the objectives for the EVMA, too. The EVM system should be able to:

\(^5\)Prior knowledge may always be introduced by shifting initial probabilities towards a particular bias. This might be especially useful when scaling to complex problems and when expert knowledge is available.
1. develop arbitrary problem-specific representations,

2. run arbitrary learning algorithms,

3. find good, problem-specific learning algorithms, as quickly as possible,

4. find algorithms for finding learning algorithms.

Conventional learning systems have a fixed learning strategy for selecting and testing solution candidates from some ‘non-universal’ search space. Incremental self-improvement, however, does not only search for solutions to some specific tasks, but also for learning strategies for finding solutions. Doesn’t the system’s universality increase its search space? In general, it does. With many toy tasks, an external user will be able to provide a conventional learning algorithm with enough problem-specific bias to solve a certain task more quickly than (initially less informed) search based on incremental self-improvement. On the other hand, however, unlike previous learning methods, incremental self-improvement can use experience to modify its search in a universal way, by exploiting arbitrary task-specific regularities if there are any, and by creating its own problem specific bias. In the long run, this advantage may outweigh initial disadvantages due to universality. (Schmidhuber, 1994).

In other words, the price to pay for the meta-learning properties and incremental learning is a bigger search space that results in a long delay at the beginning before the search effectively takes off. It is computationally demanding to run EVM-based experiments, but, given enough initial computational time, and given hard enough problems to solve, it seems plausible to assume that the EVM-based self-improving and incremental learning method should eventually outperform traditional machine learning techniques. Some evidence to support this conjecture are given in the experiments described in Chapter 9, although further studies to verify this are needed.

8.4.3 Implementation of policies

For every instruction of \( P \), there is a probability distribution over its possible values. That is, for every instruction of \( P \) there is an associated probability distribution from which that instruction is being sampled. The EVMI maintains a table of probabilities as depicted in Figure 8.2. Values for \( P \)’s instructions are selected according to their probability distribution. The meta-instructions of EVM assembly language, \( \text{inc}P \) and \( \text{dec}P \), that allow PMPs, consist in increasing or decreasing entries in that table\(^6\).

\(^6\)We need two meta-instructions, \( \text{inc}P \) and \( \text{dec}P \), which respectively increase and decrease by \( \delta \) the probability of having value \( v \) for instruction \( i \) of \( P \). \( \delta, v, \) and \( i \) are passed as arguments of \( \text{inc}P \) and \( \text{dec}P \). Note that these two instructions can be collapsed into a unique one if we allowed negative increments.
Instructions of program $p$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>0.9</td>
<td>0</td>
<td>0.1</td>
<td>0.01</td>
<td>0.2</td>
<td>0</td>
<td>0.8</td>
<td>0</td>
</tr>
<tr>
<td><strong>incP</strong></td>
<td>0.01</td>
<td>0.3</td>
<td>0.1</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>decP</strong></td>
<td>0</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.6</td>
<td>0</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>const 2</strong></td>
<td>0.05</td>
<td>0.5</td>
<td>0.04</td>
<td>0.01</td>
<td>0.1</td>
<td>0.1</td>
<td>0.05</td>
<td>0.7</td>
</tr>
<tr>
<td><strong>goto</strong></td>
<td>0.01</td>
<td>0.02</td>
<td>0.6</td>
<td>0.01</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>mul</strong></td>
<td>0</td>
<td>0.07</td>
<td>0</td>
<td>0.2</td>
<td>0.01</td>
<td>0.9</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td><strong>...</strong></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 8.2: Policies are managed by maintaining probability distributions over the possible values for $P$’s instructions. Instructions $\text{incP}$ and $\text{decP}$ modify these probability distributions. Because $\text{incP}$ and $\text{decP}$ can also modify their own probability, we speak here of meta-learning. This table is modified by the meta-instructions. At checkpoints, EIRA undoes the modifications that have not been followed by an acceleration of the rewards intake. This table, though, is never reset to its initial state.

The described mechanism expects $P$ to have fixed length. To enable variable length programs, we have introduced a special $\text{halt}$ instruction that will halt an executing program. However, $P$ may take too long to execute or it may never terminate. For practical reasons during our experiments, we always operate in a time-bound computation model. That means that after a predefined time-limit, the program is terminated. Further studies related to more elaborate estimations of program terminations will be a subject of further studies.

When a neighbour asks for a given cell’s program section 2), the EVM cell sends the program without the meta and checkpoint instructions. The outer looping instructions (if present) are also stripped out. The purpose of these special instructions is to accumulate bias information about certain instructions of the cell’s program, so as to increase their probabilities of being used. The bias mechanism works exclusively on a given cell’s program alone, and not on the one of its neighbours.

In the case of multiple programs per cell, each cell contains a single program $P$, which is able to produce a machine $m$. We have extended the original EVM design by a special data structure to represent a machine and some instructions to manipulate it (see Section 3.6 for details).

In our initial EVM implementation (Nowostawski et al., 2004), we used the list-stack and all the EVM instructions to manipulate it. The list-stack simply becomes the cell’s machine. That way, the program is able to generate machines of any dimensions (i.e. any number of programs and any number of instructions per program).

**Extending EIRA to manipulate a single EVM machine.** It is also possible to extend the original EIRA model to provide a mechanism for a single EVM machine generation. This could facilitate an environment that exploits the stochastic search method described in Section 9.9. As for the stochastic method, the machine’s dimensions are fixed and the EVM system maintains a table of probabilities.
for the possible values for every instruction in the machine. To enable the program $P$ to manipulate this table, two instructions have to be added to $P$’s language: one to increase the probability of a given value for a given instruction in $m$, and one to decrease it$^7$. All the parameters of those instructions (increment, value, and instruction) are passed as arguments through the operand stack. The experimental studies and technical details of the discussed above model are provided later in the Chapter 9.

---

$^7$These two instructions can also collapse into a unique one if we allowed negative increments.
Chapter 9

Experimental studies
The hard task is to know what the skills are that you have to replicate on machines and I think we have a huge way to go.

(Aaron Sloman)

A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away.

(Antoine de Saint-Exupery)

9.1 Introduction

In this chapter, we present results and design ideas of some of the experimental and simulation studies that have been conducted during the course of the studies on evolvable virtual machines. These experimental studies exploit different aspects of the evolvable computational architectures and provide necessary evaluation of the concrete implementations of the EVMI (see Chapter 3). Our experimentation studies include machine discovery, single-task and multi-task learning, inter-dependencies analysis between tasks, hypercycles, autopoietic dynamics (see Chapter 7), and to a certain extent, study of heuristics, random search and exhaustive search. The main overall objective is to investigate, formulate, and test hypotheses related to the main thesis, formulated in Section 1.3.

This chapter is organised, in more or less the chronological order of the actually conducted experimental studies, to give the reader a clear indication of the various stages the EVMA studies were going through. All the other chapters are organised in a linear fashion based mainly on the themes and various aspects of the studies, disregarding the actual temporal order in which the results or formulations were obtained. Therefore, some of the experiments may seem detached from the actual final EVMA, but when read in the context of the studies on evolvable virtual machines, they should give the reader a better understanding of the limitations of the existing models, and most importantly, the adaptable and evolvable aspects that the EVMA addresses.

9.2 Film analysis vs. quantitative studies

Due to the nature of the studies of complex dynamical systems and unpredictable elements when exploring an unknown state space, it can be difficult to establish simple quantities that can be measured and plotted on graphs. Even though the field of artificial life is well established in the research community, it lacks many of the methodologies of the hard sciences, and the field resists strict quantitative analysis (for which it has been criticised (Levy, 1993)). Thus it is a challenge to propose a simple yet effective methodology for studying the dynamical properties of artificial life. Within ALife itself, experimentation and observing the trajectory of the evolving and adapting dynamical system based on selected attributes seems to be the most effective and widely used technique. Notably, Wolfram
(2002) argues for the use of simulations and exhaustive search as the most effective and promising methodological approach.

In our studies, the main methodology used throughout the different experimental setups was to use simulations and “memorise” them in a sequence of visualisation frames. These sequences from the simulation were later assembled into a movie and thereby became visually amenable for analysis (off-line). The movies represent visualisation of the selected dynamics of the collection of cells. Using this approach we have run simulations and prepared 2D movies (animation sequences) of the dynamics of the evolving web of cells. Most of the observed phenomena would be difficult to be discovered by any other means. Of course, observations undertaken by this movie technique are subjective and do not represent statistically meaningful results on their own. They are, however, invaluable in identifying certain phenomena and investigated properties, and this technique proved invaluable in exploring certain unknown-in-advance dynamical patterns. Once the phenomenon is identified, then it is possible to prepare experiments and collect enough statistical data to confirm the initial observations through multiple runs and analysis. The movie technique proved to be successful in our investigations, where some of the observed phenomena were emergent and not presumed a priori.

It is important to note that our studies were not initially focused on any benchmark problems nor were intended to solve any problems in particular. We were interested in investigations of the overall properties of the EVMA, the dynamics and evolution of the system in time, and the search process under certain constraints. This approach is neither new nor unique to our EVMA studies. In the case of cellular automata (CAs), formal analysis methods are inadequate; therefore computational studies of two-dimensional static visualisations of the evolving one-dimensional CAs are used instead (Wolfram, 2002). In our case, analysis is not adequately served by formal models of computability and static two-, or three-dimensional visualisation techniques are not adequate, either. Only dynamical visualisations, such as movies in 2D- or 3D-spaces, can help to represent some of the spontaneously occurring phenomena.

Consequently, it is our belief that the utilisation of video sequences in this study of complex systems is a valuable and essential element. It is one of the ways, if not the only way, to capture complex spatio-temporal aspects of certain phenomena that cannot be predicted in advance. For 1D cellular automata the spatio-temporal aspect can be captured by a 2D image of the evolution of the single line of cells. However, for 2D cellular automata most studies must rely on video sequences to properly represent the dynamics of the spatio-temporal structures, e.g. in Conway’s Game of Life (Gardner, 1970).

223
9.3 A-grid: pre-EVMI model

9.3.1 Description

We start our investigations into self-learning systems by extending Gecow’s (Gecow, 1986; Gecow and Hoffman, 1983) model of evolution of finite-state automata. We propose a simple model of relational grid, called a-grid. The aim of this study is the analysis of simple hypercyclic architectures and emergence of second order objects (Maturana and Varela, 1980). Second order objects are certain persistent structural patterns that are not present on the base level, but appear spontaneously during the system evolution and persist for extended periods of time. In other words, structures that are not present as first-class objects on the base level of a given dynamical system appear spontaneously and persist for significantly longer periods of time than the life-cycle of the underlying dynamics would suggest – for example, structures, such as gliders and glider-guns in Conway’s Game of Life (Gardner, 1970), are second order objects that emerge from the dynamics of the underlying cellular automata cells. The rules for “glider” are not part of any of the individual cell’s rule-set, and the “glider” persists as a first class (but second order) entity within Game of Life for an extended period of time (subject to other first class objects encountered).

We use an a-grid model for the specific task of establishing a boundary (or a boundary measure) between objects. In other words, we use an a-grid to calculate a border of second level objects based on a simple producer-consumer scenario.

The general model has the following characteristics: there is a grid, and each cell on the grid has a fixed number of neighbours. For example it can be modelled on a regular 2- or 3-dimensional grid, and in the case of a 2D grid, each cell on the grid can have 8 neighbours (the adjacent cells plus the diagonal neighbours). Each cell has a vector of commodities. Each type of cell, or species, is identified by a special code vector. A code vector is a special vector divided into two sub-vectors: one sub-vector of input commodities, i.e. goods that a given species needs in order to exist, and a sub-vector of commodities a given species produces. This is represented as a simple ternary vector, with symbols 1, 0 and x. This ternary vector, code vector, represents a cell type, or species. Symbol “1” in this code vector means that a particular position in the commodities vector is a product, symbol “0” means that this position is a consumable, and “x” means that the commodity at this position is neither product nor consumable (neutral). The cell’s vector values for consumables are automatically decreased in each time step. The products levels are in a similar way automatically increased, given that all the consumables are within a special range (e.g. more than zero). The consumables are re-filled only if an appropriate producer is within a cell neighbourhood. The code vector is relatively long compared to the original number of species, so the single random mutation of the code vector does not change the species of a given object. We use a simple Hamilton distance as a function of species classification. The original species are uniformly distributed in the species space. Note that a sequence of random mutations may render the object beyond a particular species.
In the a-grid, cells are related to each other by simple producer-consumer relationship, which is spatially constrained: only cells within pre-specified distance (neighbourhood) can engage into producer-consumer relationship. Each cell, consumes commodities for its “input” vector from surrounding cells of appropriate type. In other words cells with matching complementary $0 - 1$ code vector entries. If any of the input vector values drops down below a certain threshold (typically 0), the cell dies and its place on the grid can be occupied by another cell. Different schemes can be applied here: the strongest neighbour, random neighbour, random cell, etc.

Once the system is executed/started, all cells engage into producer-consumer interactions and settle into semi-stable or stable relationships. On one level, each cell is dependent on a number of other cells. Based on that, we create a matrix of dependencies between species. To establish patters we plot both, the dependencies between individual objects, and dependencies between species. Comparing these two measures allow us to observed emergence of stable relationships between objects, and between species.

We tried to identify these relationships which are strong and constant over time, and these which are weak, and vary. We expected that properly tuned setup should demonstrate a spontaneous emergence of higher level objects, that is, groups of self-maintaining structures composed of a chain of interacting cells. This is in fact the case and we are able to show that the boundaries of these second-order objects can be identified by simple statistical analysis of relationships between lower-level entities.

### 9.3.2 Experiments

For our experimental setups we have used various producer-consumer scenarios. We have used a concept of a node that is characterised by a dependency pattern. The dependency pattern is a vector of a given specified length, the same for all nodes. The vector has randomly assigned values representing 3 possible relationships:

- **0** – this means *do not care*, and represents a neutral resource, that is, node does not produce nor consume the resource on this position of the pattern vector;

- **1** – this means *the need*, and represents a desired resource that a given node needs to function properly; the node with 1 of the pattern vector must link directly with another node that has value 2 on the same position, to consume the resource.

- **2** – this means *the product*, and represents a product that a given node produces; nodes with 2 on a given pattern position will be used by nodes that have 1 on their position accordingly.

For example, consider a simple scenario of 3 nodes, with the following patterns: $C_1 : < 1, 0, 2 >$, $C_2 : < 2, 1, 0 >$ and $C_3 : < 0, 2, 1 >$. For $C_1$ to exist it needs to attach to a node that has 2 in the first position of its pattern vector. Therefore, $C_1$ would attach to $C_2$. On the other hand, $C_2$ needs
a producer node for the second resource (the second position of the pattern vector); hence $C_2$ would
attach to $C_3$ (they have a matching $1-2$ pattern on the second position of the pattern vector. And so
on. $C_3$ would connect with $C_1$. This triangle $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_1$ is actually a simple example of
cyclic dependency. This would form a self-sustaining cycle, because all three nodes would produce
and consume each other resources forming an auto-catalytic cycle.

We use an abstract space containing a set of nodes that form various connections depending on
their consumer-producer pattern vector. We initialise all the nodes with random values of 0, 1, 2, and
we allow in each generation a single node to check a single other node, and attach to it if it pro-
vides/produces any of the resources that the given node needs. We have tried pattern lengths varying
from 3 to 1000. The results are basically the same with varying variance and standard deviations
depending on the pattern length. We have also tried varying the number of nodes in the experiments
from 10 to 1000. With our tested implementation, experimentation in the upper range of the pattern
length and node numbers was increasingly impractical, due to long computation times and visualisa-
tion problems. Most results presented here use a population of 100 nodes, with 3 or as the 100 pattern
length.

Figure 9.1: A-Grid missing links. Links statistics for a-grid runs with pattern length=100, population
size=100 and an average of 25 independent runs.

To establish how many generations we need to run the simulations for, we calculated how many
generations are needed for a simple pattern of 100 to converge so that all their requirements are

226
satisfied by linking to appropriate providers. The average pattern will have 33.33 patterns with value 1. On average, attaching to another randomly generated node should yield a 33% success rate of proper attachment with 1 – 2 pattern. Therefore, after a single iteration (generation), we should have 22.22 not attached “needs”. And so on. The theoretical requirement curve is plotted on Figure 9.1, together with stochastic experiments that we have conducted with a population of size of 400, a pattern size of 100, and 25 independent runs. The points are the average results over those 25 runs, and the error bars are calculated as three times the standard deviation. As can be seen, initially the variance is quite large, but as we converge to the all needs being connected to appropriate consumers, the variance quickly diminishes. It appears clear that the convergence happens at about 17th iteration, therefore we can expect nodes quickly forming clusters of dependencies.

To understand better the dynamics of the cluster formation, we have decided to plot the average number of links from a single node. Figure 9.2 depicts an example based on pattern length of 100, population size of 400, and 25 independent runs averaged accordingly. We have used two slightly different method of node linking. The non-trimming method attaches a new node if it has any products for the needs for which a given node does not have any producers. That means that during the random node attachment, certain nodes will have more nodes that they need (due to the redundancy of some of the products – more than one product satisfying a given node’s need). This is shown on the upper diagram. The second method is based on attaching a new node and then trimming the number of connections so as to establish only the necessary minimum of connected nodes – duplicated redundant producers are removed. This is depicted on the lower diagram. It is demonstrated that the trimming method eventually converges to the minimal number of connections, and also, throughout the process tends to have smaller variance. All these processes are based on uniform probability distributions of attachment and removal and are simulated with a random number generated populations.

Figure 9.3 depicts the result of running a simple experiment with 100 nodes over a period of 10 iterations with the trimming mechanism. As depicted on the figure, the nodes form a single giant web of inter-dependencies. When random nodes are removed and replaced with new generated nodes, they quickly assemble into the edges of this cluster. The results are independent on the number of nodes; we have tried up to 10000 with a single node pattern length of only 3 and achieved the same single hypercycle as a result. Figure 9.4 depicts the result of running 20 generations of 100 nodes with the pattern size of 3. Even though the majority of the nodes do not participate in this small cluster of interconnected nodes, they were not capable of forming an alternative self-sustaining cluster.

Figure 9.5 presents an example of a partial hypercycle forming. Throughout the studies we were unable to form a proper fully-connected hypercycle. As you can see, the left auto-catalytic cycle is being used by the right-hand side larger and more complicated cluster of nodes. The node \( n_{58} \) depends on \( n_{19} \) from the other cycle, and, \( n_{58} \) is crucial for the survival of the large cluster, because many cells depend on it. This result has been achieved by a slightly modified trimming mechanism. Beside removing redundant connections, we have also removed cells that were only consuming resources,
Figure 9.2: A-Grid established links. Links statistics for a-grid runs with pattern length=100, population size=400 and a 25 independent run average.
Figure 9.3: A-Grid example of a single giant hypercycle forming out of a 100 nodes cluster with nodes’ pattern length of 100.
without being used by any other cells as producers. We refer to those cells as parasites.

To investigate why hypercyclic linkages were not forming in our experiments, we have focused on the initial stages of the simulations, and Figure 9.6 demonstrates how, initially, many cycles and cyclic loops are formed. The upper figure shows the global state of the cell cluster after the first initial random linking, and the bottom diagram shows the state after the trimming mechanisms were deployed. It clearly demonstrates that many small loops and cycles form initially; however, only one eventually dominates and takes over the entire node space, forming always a single giant complex node cluster, with complex dependency patterns (as shown earlier on Figure 9.3).

9.3.3 Discussion of the results

We have observed that the producer-consumer interactions are stable up to the first order of hypercyclic dependency. However, more complex stable chains of dependencies were not attainable by chance or by random initialisations and random attachment schemes. This is consistent with other research in the area of artificial chemistries, for example (Dittrich et al., 1998), who also demonstrated short spontaneous hypercyclic dependency chains. It seems from our experiments and other studies, that the spontaneous appearance of short hypercyclic dependencies in such a random soup is relatively unlikely, and additional mechanisms for aggregation and spatial management of nodes must be provided through external environmental pressures. Our desire was that the properly tuned setup should demonstrate a spontaneous emergence of higher level objects, that is, groups of self-maintaining structures composed of a chain of interacting nodes. This was only achieved to a certain level. The a-grid experiments demonstrated that further investigations into the detailed mechanisms of aggregation of hypercyclic dependencies must be conducted and that external mechanisms that stimulate higher-order objects emergence and maintenance need to be provided. A-grid was the first empirical indication for the steps leading to the development of the EVMA.

9.4 Symbolic regression: preliminary studies

9.4.1 Description

During the course of designing and developing the EVMA, we have investigated various EVMI and implementation strategies. One of the preliminary attempts was to re-use one of the existing virtual machines and perform direct bytecode manipulation during runtime to achieve desired dynamical behaviour on the VM level. For this purpose, we have used a simple subset of Java Virtual Machine (Venners, 1999) bytecode and run some tests on a set of simple symbolic regression tests using our custom-built genetic algorithm system. We used the developed framework for testing the model on simple symbolic regression problems. It was not our intention to compare it with other symbolic regression frameworks, hence we have not performed any comparative studies on this problem. Our goal was to test the proof-of-concept framework of a self-modifiable (on the bytecode level) virtual
Figure 9.4: A-Grid example of a hypercycle formed after 20 generations of 100 nodes with a pattern length of 3.

Figure 9.5: A hypercycle formed after 20 generations of 100 nodes with a pattern length of 100. This was achieved through the trimming mechanism and removing parasites.
Figure 9.6: (A) an example of initial links and node cluster forming. Example after first iteration before trimming, 100 nodes with pattern length of 100.

Figure 9.7: (B) an example of initial links and node cluster forming. Example after first iteration after trimming, 100 nodes with pattern length of 100.
We have tested the idea on the Java Virtual machine 1.3, with the use of the BCEL bytecode manipulation library (Project, 2000). BCEL, the Byte Code Engineering Library is intended to give users a convenient on-the-fly and on-line possibility to analyse, create, and manipulate Java classes. Classes (types) are represented by first-class objects within JVM. They contain all the symbolic information of the given class: methods, fields and bytecode instructions. Users can freely manipulate and manage all the class internal data-structures and freely modify the bytecode of the method definitions. Such objects can be transformed by a program and re-loaded into memory – for example, with a modified implementation of certain methods. The actual definitions of the bytecode instructions themselves cannot be changed. Hence, the initial idea was to develop an intermediate VM that operates on “shadow copies” of the real JVM bytecode instructions, allowing the user to redefine the copies.

9.4.2 Experimental setup

For the proof-of-concept setup, we have used a small subset of the full JVM instruction set, namely all the double-precision arithmetic operators (DMUL, DDIV, DADD, DSUB), together with a helper stack loading instruction (DLOAD_1), which loads a first variable passed to the method under generation onto the JVM operand stack. We have tested the framework on some toy problems from the symbolic regression class. The results below are for the problem with a single variable $x$ and fitting a target equation $x + x^2 + x^3 + x^4$. The solution candidates were tested on an array of 100 randomly generated values, and a root mean square error was used to measure how well a candidate equation fit into the target equation. We used a combination of tuning parameters to estimate the performance of the framework and of the search of the program space with the use of a stack. A typical run would look as follows:

- Population of 15 individuals, length of the bytecode programs fixed at length 14, roulette-wheel selection, Montana-Davis crossover with probability 0.8, random change of the whole genome with probability 0.3, uniform on all genes. On a Pentium 4 processor (2.8MHz) this was 66060 milliseconds. Below is a bytecode dump of the discovered solution after 247 iterations:

```
.method public process(D)D
.limit stack 50
.limit locals 3
.var 0 is this Lorg/rakiura/cirrus/bytecode/ProcessorAdapter; \
 from Label0 to Label1
.var 1 is arg0 D from Label0 to Label1
.var 0 is this Lorg/rakiura/cirrus/bytecode/ProcessorAdapter; \
 from Label0 to Label5
.var 1 is argument D from Label0 to Label5
Label0:
.line 38
```
This program has 14 instructions, 0 nops, with 4 useless instructions: the first 3 instructions are \([\text{dload}_1 \text{ dmul dup}]\) and the 5th instruction \text{dup2}. The program does the following: 
\[(x^2 \cdot x + x) + (x^2 \cdot x + x) \cdot x.\]

- A different correct solution, found after 161 iterations in 38338 milliseconds.

```
.method public process(D)D
.limit stack 50
.limit locals 3
.var 0 is this Lorg/rakiura/cirrus/bytecode/ProcessorAdapter; \  from Label0 to Label1
.var 1 is arg0 D from Label0 to Label1
.var 0 is this Lorg/rakiura/cirrus/bytecode/ProcessorAdapter; \  from Label0 to Label5
.var 1 is argument D from Label0 to Label5
Label0:
.line 38
dload_1
Label5:
dload_1
nop
nop
dmul
```
This program has 11 instructions plus 3 nops, with one useless instruction (the first `dup2`). The program performs the following: \((x^2 + x) + (x^2 + x) \cdot x \cdot x\).

Note, the developed framework manipulates and generates bytecode operations only between `Label5` and `Label1` in the above bytecode dumps. The initial loading of the parameter onto the stack and returning the final result from the stack through the `dreturn` bytecode operation is part of the fixed code block of the framework itself.

The main focus of conducting these tests was two-fold:

1. to establish preliminary intuitions and patterns of behaviour of a stack-based stochastic code-generation framework;

2. to establish the feasibility of using the JVM and Java bytecode directly for the purpose of building the EVMI.

As shown on the two previous examples, the use of the JVM operand stack for the purpose of code generation is possible. It is characterised by properties slightly different from a simple machine code generators. Due to the nature of stack-based processing, the discovered code must make a rather complicated reuse of previously established structures to build a complete correct solution to a given problem. Note in the previously discussed examples the reuse of large coding portions \((x^2 \cdot x + x)\) and \((x^2 + x)\) respectively.

Most of the other correct solutions found for this symbolic regression problem followed one or the other coding scheme with a different order or combination of the `dup2` and `dload_1` instructions (as in the first case below). But, there were some other unusual code re-use examples:

- `dload_1``dmul``dup2`
dload_1
dload_1
dup2
dup2
dmul
dmul
dadd
dup2
dload_1
dmul
dadd

Instructions: 14, nops: 0, useless instructions: 4, coding: \((x^2 + x) + (x^2 + x) \cdot x \cdot x\).

• dload_1
dadd
dload_1
dup2
dload_1
dmul
dload_1
dmul
dadd
dup2
dload_1
dmul
dadd
nop

Instructions: 13, nops: 1, useless instructions: 4, coding: \((x^2 \cdot x + x) + (x^2 \cdot x + x) \cdot x\). Note, this code achieves the same result as the previous solution \([\text{dload}_1 \ \text{dmul} \ \text{dup2} \ \text{dload}_1 \ \text{dup2} \ \text{dload}_1 \ \text{dmul} \ \text{dmul} \ \text{dadd} \ \text{dup2} \ \text{dload}_1 \ \text{dmul} \ \text{dadd}]\), but uses a slightly different path.

• dload_1
dload_1
dup2
ddiv
dload_1
dload_1
dmul
dadd
nop
dmul
dup2
dload_1
dmul
dadd
ddadd

dmul
dadd
nop
dmul
dup2
dload_1
dmul
ddadd
ddadd

dmul
dadd
nop
dmul
ddadd
dload_1
dmul

dmul
dmul
dload_1
dmul
dload_1
dmul
dmul
dload_1
dload_1
dmul
dmul
dload_1
dadd

Instructions: 13, nops: 1, useless instructions: 0, coding: \((x^2 + x) \cdot x + (x^2 + x) \cdot x \cdot x\). Note this was the only solution that made no useless stack manipulations at all, and left only a single correct answer on top of the empty stack at the end of the execution\(^1\).

There were a number of partial solutions that dominated the population for a majority of the iterations waiting for one or two useful mutations. The partial solutions we have observed can be grouped into three classes:

- \(\text{dload}_1 \ \text{dup2} \ \text{dload}_1 \ \text{dmul} \ \text{dload}_1 \ \text{dmul} \ \text{dadd} \ \text{dup2} \ \text{dup2}, \text{coding: } (x^2 \cdot x) + (x^2 \cdot x) \cdot x\).
- \(\text{dup2} \ \text{nop} \ \text{dup2} \ \text{dmul} \ \text{dload}_1 \ \text{dload}_1 \ \text{dload}_1 \ \text{dmul} \ \text{dload}_1 \ \text{dload}_1 \ \text{dload}_1 \ \text{dmul} \ \text{dload}_1 \ \text{dadd} \ \text{dmul} \ \text{dmul}, \text{coding: } (x^2 + x) \cdot x \cdot x\).
- \(\text{dload}_1 \ \text{dup2} \ \text{dmul} \ \text{dload}_1 \ \text{dload}_1 \ \text{dmul} \ \text{dload}_1 \ \text{dload}_1 \ \text{nop} \ \text{dadd} \ \text{nop} \ \text{nop} \ \text{dmul} \ \text{dload}_1 \ \text{dadd}, \text{coding: } (x^2 + x) \cdot x^2 + x\).

\section*{9.4.3 Results and discussion}

The main objectives for the JVM-based symbolic regression experiments were to establish suitability of the bytecode manipulation within the JVM\(^2\) and to investigate properties of stack-based code generation.

\textbf{Suitability of the JVM direct bytecode manipulation.} The suitability of the JVM for a framework based on direct bytecode generation/manipulation is severely limited by two factors:

1. Lack of garbage collection or memory release of the class-related data-structures.

2. Strict code verification and error checking. Invalid operand stack instructions always result in a segmentation faults and a crash of the entire VM.

\(^1\)It would be rather challenging for a human being to propose such a solution scheme in a reasonable time given only pen and paper.

\(^2\)JVM version 1.3.1 at the time of testing.
Due to the static nature of the class data-structures within the JVM itself, long experiments with a large number of individuals, or life-long reinforcement learning were not feasible with the investigated scheme. The second major limiting factor, apart from the lack of garbage collection of static class data structures, is the inability to load and execute not-properly-aligned or faulty code fragments. The JVM, upon loading of class data (e.g. when a new method body has been dynamically generated), conducts a process called verification, which checks if the method structure, stack usage, and all instructions are correctly generated. For a majority of code segments that are generated in a purely random fashion, verification fails and the JVM itself halts. There is no ability to continue execution with erroneous code segments in the JVM\(^3\).

In summary, for simple evolutionary or stochastic code generation mechanisms, the framework based on direct Java bytecode manipulation works well and can be used successfully for various Java-based automated code generation tasks. The use of the BCEL library simplifies all major tasks of processing the bytecode and generating/loading the dynamically created methods/classes into the running instance of the JVM. However, for complex systems where code generation is the primary objective and life-long reinforcement learning is a dominant feature, the JVM itself is prohibitively restrictive. A different way of implementing EVMI must be formulated.

To achieve a set of properties that our EVMI needs to exhibit, we found it necessary to re-implemented a large subset of a virtual machine based on the JVM design, ourselves. Our initial implementation was developed in Java. See Chapter 3 for detailed description of the implementation.

9.5 Machine discovery

The above described experiments might be considered as preliminary work leading to the formulation of the EVMA. The main objective for conducting these earlier experiments was to test certain capabilities and the feasibility of particular reification mechanisms and implementations of reflective capabilities on the VM level. Thanks to the results of these early investigations we gained insights and intuitions of how a self-reflective, self-re-definable VM implementation can be potentially engineered. Based on these intuitions, we designed the preliminary EVMI and the EVM assembly language, developing at the same time the abstract architecture that was later formalised in the form of the EVMA. All the initial experimental studies were conducted on simplified models without the use of the actual EVMI, as such. All the experiments described in this and following sections make use of the EVM framework and the EVM language. It should be noted that the EVM language, together with the actual framework implementation, were iteratively refined and re-implemented. However, all the main (abstract) design objectives were always constant and are common across all those different experimental setups. These form the core of the EVMA as it is presented in Chapter 2.

The process of machine discovery is central to the EVMA search process. Machine here means

---

\(^3\)Note, that these facilities are provided in the EVM through the special instructions \texttt{coo} and \texttt{modulo-rule}.
simply a list of subprograms that can be utilised to solve a particular class of problems. The concept of a *machine* in the EVMA is somewhat “overloaded”. A single machine can be implemented and used in various ways within a single EVMI.

- A collection of EVM cells can be seen as a single *machine*.
- A single EVM cell can contain a single program, or it can contain multiple programs, which can be seen as a *machine*.
- A single EVM program can create, manipulate and use multiple machines on various levels.

The EVM assembly language itself is designed around the concept of a machine (details provided in Chapter 3). Programs can dynamically create, manipulate and use machines within a single cell. Note, a single program can express the structure and execution flow of computations conducted on multiple EVM cells. For some experiments we have implemented the cell as containing several programs (we will refer to these programs simply as a single *machine*). In these experiments the primary goal is to specialise a single cell in solving a set of tasks by discovering a set of programs constituting a single computational machine.

The term *complexity* of a program (or a cell) is used here in analogy to *algorithmic complexity* (Li and Vitányi, 1997); see also Section 4.8.4 for detailed discussion on the concept of algorithmic complexity. We use the length of the shortest program exhibiting certain behaviour as a measure of program (or task) complexity. See also the detailed discussion in Section 2.9. The complexity of a given discovered machine (a cell, or a collection of cells) depends on the task at hand. That means the task that cells try to solve represents the ultimate complexity level that the cell can achieve.

In highly dynamic environments, the cells may not develop into higher complexity levels (with multiple programs), because it would be (a) too time-consuming to do so, and (b) the dynamic environment puts a time limit of how much can be *discovered* for a given task before the environment changes and so do the target tasks. However, in static environments, the cells may have sufficient amount of time to try to benefit from more than a single goal (task solving), and it is possible to develop higher complexity levels within a single cell. In other words, in static environments it is easier to explore the search space and discover patterns and correlations.

It is our working hypothesis that in evolutionary terms, the main drive of increased complexity is directly related to the stability of the environment. We hold this view despite the common belief that complexity is simply the function of evolutionary time: organisms evolved later being more complex to previously developed organisms. Such a view does not provide any explanatory value in terms of complexity and computational evolution. Note that these two hypotheses overlap, because early organisms like bacteria are faced with highly dynamic and unpredictable environments, whereas later, so called higher-order organisms are usually more complex, but function in more static and predictable environments (homeostasis, and autopoietic hypercycles). Certain cells in biological organisms develop, function and die in exactly the same environments today, as they did millions of
years ago. The more stable the environment, the more complex the organisms can evolve as they can establish long-lasting static relationships with various aspects of their own “environments”. The more stable the environment, the larger the number of combined goals a given organism can try to solve or adapt to. In highly dynamic environments only simple and short solutions will evolve. We mean here short as in the algorithmic complexity measure, see Section 2.9 for details. In analogy, primitive bacteria exposed to highly dynamic environments are very robust and flexible, but, are relatively simple life-forms. On the other hand, large-collections of cells in higher-species form highly complex networks of dependencies, where the stability of the neighbourhood and homeostasis of the processes are central to the existence of the complex life-forms.

To investigate the working hypothesis of the stability of the environment, a number of experiments have been conducted. Some of these experiments will be discussed later in this chapter.

9.6 Virtual ecosystem

Figure 9.8: Toroidal space. Each cell on a torus has the same neighbours as if an “infinite” two-dimensional space was used. A finite two-dimensional space is wrapped around, in a donut-like shape, without any beginning or end.

9.6.1 Description

These experiments are the first series of EVMI experiments conducted with a regular grid of interacting programs. The design of the experiments is to investigate the inter-relations between instruction sequences in a regular 2D grid, with a resource sharing and multi-task learning environment. One of the prevailing feature of this class of experiments is the study of the emergence and life-cycle of parasite and parasitic relationships between computational cells.

In our experiments we have used a regular toroidal grid (see Figure 9.8), with only one program per cell. We have used a 4-neighbour model (up, down, right, left). The locality of the interactions
plays an important role in the dynamical evolution of the cells' interactions. Our cellular system demonstrates certain artificial life features. For instance, as knowledge diffusion is not as easy as with several programs per machine, the system has to find some artifacts (tricks) to propagate solutions to the cells that need it.

These experiments are conducted on a single level of EVM, with the aim of discovering an effective compact machine suitable for a given sequence of tasks. We have used two probability-based learning methods, both based on biased random search. That is, the search is non-deterministic, but based on pre-allocated and modifiable probability distributions of instructions. We use two variations: one based on manipulating probabilities of individual instructions and another model based on conditional probabilities of sequences of instructions.

In particular, we have introduced, investigated and compared five different specialisation mechanisms:

1. Basic random search and exhaustive search.
2. Genetic algorithms-based search.
3. Ad-hoc stochastic search for a fixed-size program.
4. Improved tree-based stochastic search.
5. Universal reinforcement acceleration mechanism.

All these search mechanisms operate at the low, cellular level and dictate, in interaction with the environment (for detailed description of EVM instantiation used see Chapter 3), the global, macroscopic behaviour of the system.

An example run, exhibiting some typical features of our self-organising, self-adaptable cellular system is presented in Figure 9.9.

In order to be efficient in a multitask context, we generally assume that a model should fulfill the following requirements:

1. All tasks must be eventually solved.
2. Solving difficult tasks should lead to greater rewards than solving easy ones (solving difficult and long tasks may consume significantly more resources than solving easy/short tasks).
3. Computational resources should be primarily allocated to efforts dedicated to solving all the as yet unsolved tasks.
4. Solutions to solved tasks must not be forgotten as long as they are useful.

---

4By *knowledge diffusion* we mean the ability to propagate and spread solutions discovered in one part of the grid to other parts of the grid. This ability helps knowledge re-use and speeds up the discovery of complex solutions.
iteration 3
No solution found yet.

iteration 15
First solutions found for $2x$ and $3x$.

iteration 75
More solutions have been found. Some involve several cells (with white cells). Also some parasites (smaller cells).

iteration 165
Competition: More and more cells -> less and less food -> only the shorter solutions survive (don’t need to share their food).

iteration 320
Catastrophe: most of the yellow cells have disappear.

iteration 2650
Notice that solutions are not always at the same placeholders. Because they appear and disappear potentially everywhere, it ensures that some cells have useful neighbours.

iteration 3200
Eventually, a cell reuses the solutions in its neighbourhood to solve $3x+2y$. Actually, it needs fourth (white) cell to access the green solution. Together, these 4 cells (red-white-green-yellow) form a self-assembly block.

iteration 2350
Knowledge diffusion: shortly after, a cluster of parasites appear around the solution.

iteration 4100
Notice the dynamism on the grid: most solutions are at different locations. However, the block computing $3x+2y$ is extremely stable, since it gains a lot of rewards.

iteration 11950
Good solutions are not forgotten: Catastrophes don’t affect the block computing $3x+2y$.

iteration 15000
A new (shorter) solution for $3x+2y$ appears.

iteration 15450
The new solution starts diffusing with parasites. The first red cluster diminishes owing to the competition for the food with the new solutions.

iteration 22600
One more cluster for $3x+2y$. Not affected by catastrophes.

iteration 27000
More clusters. Less parasites. In the long run, parasites will disappear.

iteration 27000
More clusters. Less parasites. In the long run, parasites will disappear.

iteration 50289
Adaptability: The task $2x$ has been removed. All the cells that used to solve it forget the solution, except the still useful ones (in white) that are reused to compute $3x+2y$.

Figure 9.9: Typical evolution of a cellular system. Note that the topology is toroidal. For example, right neighbours of the cells on the right border of the grid are the cells along the left border of the grid.
5. Knowledge diffusion should be facilitated automatically, i.e. solutions to previously solved tasks must be available to all the efforts trying to solve new tasks.

6. Dynamic environments should be supported: tasks can be added and/or removed at any time, dynamically.

More details pertinent to the arithmetical experiments are provided later in Section 9.7.3.

9.6.2 Cellular topology

For the following set of experiments our cellular system is organised into a web of interacting cells. Time and space are discrete. All cells evolve in parallel in an asynchronous way, continuously trying to solve tasks by stochastic exploration of alternative configurations. A cell can try to solve a task on its own, and it can also reuse its neighbours’ programs at will (the neighbours programs are part of the dynamic instruction set of the cell).

When a cell receives a positive reinforcement (a reward), the reward is shared with all cells that were used to compute the solution. All of the cells benefit proportionally from their symbiotic relationships. The ability to access neighbours’ programs facilitates and stimulates complex hierarchical organisation, i.e. self-assembly. Through self-assembly individual cells are able to collaborate to solve more complex tasks.

In these experiments we have mainly focused on regular grids, because of the ability to observe the dynamics of the interacting cells. It is easy to visualise on the 2D grids the dynamics of collaborating cells. We have used a film-based visualisation technique to observe the pattern formation and dynamics. We only considered experiments on a toroidal grid of cells (see Figure 9.8) with the von Neumann neighbourhood (four neighbours: up, down, left, right). Experiments with various other topologies and with various neighbours could be performed based on the same basic framework. Various artificial topological gradients, or discontinuities (e.g. simulating an impassable obstacles in the connectivity of cells) to isolate a population of cells, are possible, but have not been thoroughly tested to date. Another interesting feature would be to experiment with a full connectivity of cells, such that every cell can access any other cell on the web. However, due to computational constraints, such experiments with large grids proved to be impractical and difficult to conduct with the currently available resources.

In our opinion, an arbitrary connectivity, but with a limited number of neighbours, is probably the most powerful realisable and universal solution. Consider a case where every cell has four neighbours but can dynamically choose them from anywhere within the web. The result is an appealing dynamic topology with relatively expressive capabilities. To some extent, since changing neighbourhoods is equivalent to a move, it could be interpreted as (but is not restricted to) a non-homogenous swarm of agents. This and related aspects of the computational EVMA have been discussed in the Chapter 4.
9.6.3 Complexity of tasks

An important question is about the nature and the complexity of the problems tackled when doing experiments. On one hand, these problems must be complex enough to fully exhibit the system’s properties and exploit its capabilities. On the other hand, they must stay sufficiently simple to fit with the computational resources available when conducting the experiments. We decided to experiment with the smallest problems exhibiting certain expected behaviour. With more complex examples we would risk an exponential increase in waiting time. As a practical matter there is no point waiting for a long time to obtain experimental results, and one could always argue that our experiments were too simple, anyway. We have tried to set up the simplest experiments that are able to exhibit the desired system’s features.

We have designed two types of experiments: maze experiments and arithmetical experiments. The first ones provide a smoother fitness function (rewards spread between 0 and 100), thereby enabling easier ability to compare the behaviour of the cell’s specialisation mechanisms. The second ones have an extremely rough fitness landscape (all-or-nothing), but they facilitate the formulation of incremental and related task spaces. We use them to investigate the behaviour of the web and co-operation between cells.

9.6.4 Program discovery within a cell

The specialisation mechanism modifies a program within a cell. It aims to find a program that yields as many rewards as possible to the particular cell given a set of tasks from an environment. The specialisation mechanism works in discrete time units, and within each time unit a cell performs a finite number of steps to produce a program. The program can be randomly generated, based on the acquired bias (history-dependent code generation), or it can be computed by the program itself (meta-learning). To recall from Chapter 3, the task manager provides the tasks to all cells. The EVM execution engine is used to run the program, and the reward mechanism is used to accumulate the rewards.

The EVM execution engine for these experiments was here adapted to support the instructions that refer to four cell’s neighbours in order to be able to execute neighbours’ programs on request. The called cell in this situation receives a program and particular input and produces an output.

9.6.5 Exhaustive search

Despite the fact that exhaustive search is not considered practical even for moderate search spaces, it also suffers from another, more fundamental problem: the dynamic nature of the environment renders exhaustive search inapplicable in general. Consider a case where an already checked code was snipped and rendered useless in time $t_1$ becomes a perfect solution at time $t_2$ where $t_1 < t_2$. In a general sense, the environment changes all the time: neighbours are evolving in parallel; some tasks
Figure 9.10: Every cell continuously performs iterations. An iteration consists in selecting a program, running it, and, depending on the rewards, modifying its internal structure. At every iteration, the execution engine tries to solve some tasks in the environment. For each task, it runs the cell’s program (possibly calling some neighbours’ programs) in interaction with the task’s resource. If the output is correct, the cell receives some rewards from the resource. The specialisation mechanism uses these rewards to modify its state and provides a new program for the next step.

It is important to understand that the “search” here means a search for a mechanism that is capable of exploiting existing regularities in the problem space, whatever they may be. If there are no regularities to be exploited, the meta-search mechanisms will not perform better than simple random search – in fact they will always perform worse. However, such problematic spaces are often impractical to assume. The existence of regularities in engineering and computational problems is often the case, and it may be claimed on philosophical grounds that such regularities are intrinsic to the way our physical universe is structured. We will leave these general discussions open, and assume that there exist unknown a priori regularities of some type in the search space.
9.7 Cell specialisation mechanisms

9.7.1 Objectives

The main objective for cell specialisation mechanisms is to investigate how to design the behaviour at the low, cellular level, to obtain the right global behaviour of the web of interacting cells. The EVM cellular system has no global control at the macroscopic level. Every individual cell operates autonomously, and cells are not influenced by the macroscopic behaviour of the web directly. However, the macroscopic performance influences the reward mechanisms, and therefore this indirectly influences individual cells.

Cell specialisation is the key part of these experiments, and this component is responsible to regulate a single cell’s behaviour. Specialisation is the process of finding a good program for the cell, i.e. a program which yields enough rewards for the cell to survive (Figure 9.10).

Given an initial set of instructions, the specialisation mechanism aims to find a program\(^5\) that solves tasks provided by the task manager. The specialisation mechanism can try several different sequences for the program of the cell. Using a feedback mechanism, the reward system, from the environment, it inevitably converges towards an appropriate sequence that yields the maximum number of rewards per time interval. In our experiments, each cell maintains a set of potential programs \(P\), and provides a single program \(p\) to be evaluated by the environment. Note the distinction between the cell’s program set \(P\) and its value \(p\) at a particular moment. \(p\) is also called the selected program for \(P\).

The specialisation mechanism provides a selected program \(p\) for \(P\) (a) at every time step to run the execution engine to evaluate the cell (and collect potential rewards), and (b) on request, when neighbours’ programs need it.

Parameters. The choice of instruction set, the choice of tasks and sub-tasks, and the actual parameters of the specialisation mechanism are important in guiding the search process. In other words, explicitly and implicitly all these parameters bias the search process. A particular parameters configuration may be well-suited for a given problem, but may be extremely inefficient for another one. We conduct our investigations with a minimal initial bias and with a minimal number of parameters, in order to investigate the capabilities of meta-mechanisms to cope with a wide range of problems. Usually, a mechanism with as few parameters as possible is computationally expensive on single, individual problems, but performs well on a wider range of problems.

Adaptability plays an important role – the specialisation mechanism has to be flexible and take advantage of changing circumstances. If the environment changes (new tasks added, tasks removed, rewards changed) the cell must adapt by possibly forgetting its current specialisation and search for new ways of solving new tasks or collecting new rewards. The change in the environment can be

\(^5\)Or several if a cell can have several programs. For every method, we first present the case with one single program per cell, and then, in a special paragraph, explain how we extend it to the case with several programs.
triggered by external factors outside of the EVM Universe, or it can be triggered by events within the environment itself. Often, the reward structure is changed after a single cell (or cluster of cells) finds the answer to a given problem – there can be non-linear reward rescaling situations which adapting cells need to react to. Cells that cannot achieve a quick adaptation are automatically eliminated from the ecosystem, based on the reward provisioning and the cell’s energy expenditure (the rewards collected by the cell dissipate automatically during the course of the cell’s computation, and once on level zero, the cell is re-initialised).

**Accumulating bias.** In addition to finding the solution to a particular task, the specialisation mechanism tries to *remember* solutions to previously solved tasks. In some specialisation mechanisms this is achieved symbolically by remembering the exact subroutines. In other mechanisms, this is achieved with the probabilistic models through shifting probability biases of certain instructions or subroutines within the program/cell structure. The intent is to store previously useful solutions within the cell’s programs or within the cell’s search biases. For instance, a cell $C$ can specialise in solving arithmetical operations if the environment provides several arithmetical problems, $C$ will more or less randomly explore the space of possible programs until it starts finding the first solutions. Then by solving the same tasks again and again, probabilities of useful arithmetic instructions will increase and successful solutions will progressively be stored in $C$’s internal structures. For the exclusively arithmetic problem solving, cells will specialise in arithmetic operations and will become more and more efficient in solving the tasks in that domain. On the other hand, they will also become less and less apt for solving any other problems.

**Rewards scaling.** The way rewards are given is important, and it is controlled by the subsystem of the environment called the *reward manager*. The rewards scaling problem is also known in traditional genetic algorithms as *fitness function tuning*: how to map the programs’ quality to rewards’ quantity. It requires a few more parameters for the environment, thus inevitably leading to a bias of the search.

Because our aim is to investigate architectures that operate initially in a bias-free environment, we propose a novel dynamic scaling mechanism. The proposed and tested *reward scaling* works dynamically and is domain independent. A cell remembers what range of rewards it used to achieve (average reward intake). Therefore it is easy to know if the current reward intake is above or below the average. Implementing this extra mechanism requires a value storage and averaging mechanism in every cell. This is modelled by a single variable, $\hat{r}$, for the *typical reward intake*. The reward scaling depends on a parameter $\gamma$, which needs to be tuned during the experiments. The reward level is maintained using the following formula\(^6\):

$$\hat{r}_{k+1} = \gamma \hat{r}_k + (1 - \gamma)r,$$

Other formulae are possible. The one used in our experiments has the advantage of being simple and efficient in terms of memory and computational time.

247
where \( r \) is the reward provided by the environment, and \( k \) is the time step. The reward effectively used by the specialisation mechanism is then \( r - \hat{r}_k \), instead of \( r \). High values for \( \gamma \) imply that we remember quite far in the past, whereas low values imply that recent rewards (especially the last one) are much more important. \( \gamma \) is typically in the range \([0.9, 0.99]\).

In the following sections four different search methods for the specialisation mechanism are presented:

1. random search,
2. genetic algorithm (GA) with variable lengths of chromosomes,
3. stochastic search based on a probability distribution of individual instructions,
4. stochastic search based on a probability tree.

The maze experiments (explained below) have been used to compare their behaviours. In order to focus on the specialisation process, we ran them using one single cell only.

### 9.7.2 Maze experiments

![Figure 9.11: Grid configuration for experiments 1, 2, and 3 respectively (the actual size of the grid is 100x100 toroidal, instead of 7x7 as depicted schematically). Green cells contain rewards.](image)

For preliminary testing of different search mechanisms, we used programs that control an agent moving on a two-dimensional discrete grid. Some of the cells on the grid contain rewards. We have also implemented the possibility of having punishments (negative rewards) and obstacles (non-crossable regions of the maze). Grids for maze experiments 1, 2, and 3 are depicted in Figure 9.11. In maze experiments 1 and 2 the rewards are persistent on the grid. In maze experiment 3, the agent can obtain each reward only once from a given cell (i.e. volatile rewards). This constraint has been added to force the agent to follow a path. The initial position of the agent is always the top left corner. Execution time is limited to 100 time steps for the first two experiments. This means, the ideal program for experiment 1 would be 100 instructions long, and will collect 100 reward units. For maze experiment 2, the perfect program would be again 100 instructions long, and would collect 99
reward units. Note, that there is an inherent trade-off between the total number of rewards and the length of the program. The longer programs are not easily discovered, but, they yield higher rewards. For a program of length 6 that utilises a loop, the total number of rewards for maze experiment 2 is 68. For the third experiment the time limit is set to 12 time steps, and the total number of possible reward units collected is 12. Note that the base instruction set of the EVMI system has been extended here with 4 special instructions: down, up, right, and left to move the agent on the grid. Table 9.1 presents some examples of possible solutions.

Table 9.1: Some good solutions for maze experiments.

<table>
<thead>
<tr>
<th>Maze n</th>
<th>Solution</th>
<th>Rewards</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>right left const_0 goto</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>right left right left const_0 goto</td>
<td>67</td>
</tr>
<tr>
<td>1</td>
<td>right left right left right left left left (100 instr)</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>down right down up const_2 goto</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>down right down right up left const_2 goto</td>
<td>68</td>
</tr>
<tr>
<td>2</td>
<td>down right down right up left down right up left const_2 goto</td>
<td>82</td>
</tr>
<tr>
<td>2</td>
<td>down right down up down up down up up ...</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>right down right down down down right right right up up left</td>
<td>12</td>
</tr>
</tbody>
</table>

### 9.7.3 Arithmetical experiments

For arithmetical problems, cell’s programs are short (typically 3-8 instructions). The reason is that we want to focus on the way these programs will collaborate and re-use other existing sub-expressions from neighbours. Some of the tasks tackled are enumerated in Table 9.2. Note that, for instance, the solution to \(2x + 3y\) can be only five instructions long (leftNeighbourProgram swap rightNeighbourProgram add halt), and thus is much more likely to be found, if a cell
Figure 9.12: Self-assembly. **Bottom.** Left: first, some cells discover the solutions to the easy tasks ($2x$ and $3x$). These solutions can be reused by their neighbour to compute a more difficult, but related task ($3x + 2y$). These three cells live in symbiosis together (middle). At the same time, two other cells (green and grey) manage to discover together a solution to ($49 - x$). The green cells compute that solution with the help of the grey cell. The grey cell solves no task alone, but still gets rewards and survives, since it contributes to the computation of the green cell. Right: Eventually, a cell between these two blocks of cells connects them to solve a more complex task ($49 - (3x + 2y)$). **Top.** Details of the cells’ programs.
has a (left) neighbour that solves \(2x\) and another (right) that solves \(3x\). That’s the whole point of our system: *knowledge reuse*.

Table 9.2: Some examples of the arithmetical tasks tackled.

<table>
<thead>
<tr>
<th>Task</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x + y)</td>
<td>add halt</td>
</tr>
<tr>
<td>(xy)</td>
<td>mul halt</td>
</tr>
<tr>
<td>(2x)</td>
<td>const(_2) mul halt</td>
</tr>
<tr>
<td>(3x)</td>
<td>const(_2) inc mul halt</td>
</tr>
<tr>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>(2x + y)</td>
<td>const(_2) mul add halt</td>
</tr>
<tr>
<td>(2x - y)</td>
<td>const(_2) mul sub halt</td>
</tr>
<tr>
<td>(2x + 3y)</td>
<td>const(_2) mul swap const(_2) inc mul add halt</td>
</tr>
<tr>
<td>7</td>
<td>const(_2) const(_2) inc add halt</td>
</tr>
</tbody>
</table>

### 9.8 Random Search mechanism

#### 9.8.1 Description

Random search is the simplest mechanism to use in order to specialise a cell. It has two states (see Figure 9.13). In the first, initial state, it randomly generates a new program set \(P\) for a given cell and selects a single program \(p\) from the set. If a selected program \(p\) is rewarded, the mechanism transits to the second state. In the second state, \(p\) is always selected for a given cell. The cell will stay (*survive*) in that state as long as \(p\) remains successful. Note that due to the dynamic nature of the environment, once the environment changes, the selected \(p\) may become inapplicable and stop yielding rewards. The mechanism is implemented by storing the cumulative rewards (called *provisions*) gained by \(p\). At every time step, a fixed amount \(F_{NEEDED}\) is subtracted from *provisions*. If the cumulative reward value drops below zero, \(p\) is considered unsuccessful, and the mechanism transits back to state 1. In other words, the cell is re-initialised.

![Figure 9.13: The random search mechanism.](image-url)
9.8.2 Observations

On one hand, random search cannot take advantage of regularities in the fitness landscape. But on the other hand, it has no parameters, is extremely fast and needs very little memory and computational resources.

In the general case, random-search-based mechanisms tend to get stuck in a sub-optimal solutions as long as the solutions bring any reward. This, to some extent, would be true for our search too, if we tried to specialise one single cell only. However, because we use a massive collection of many competing cells in parallel, this is not anymore a problem. Cells compete implicitly between each other, and only the ones that get more rewards will have enough food to persist on the grid. That competitive pressure ensures that cells always try to find better solutions, and the random-search continuously tries to escape local optima.

For our tests with arithmetical problems, the landscape basically consists of one big peak with a steep slope (rewards are either all or nothing). In such circumstances, random search is appropriate and performs quite well when compared with other methods. Moreover, by implementing the simplest possible search mechanism for every cell, it is possible to focus the observations on macroscopic behavioural patterns, i.e. how cells interact to compute a complex solution.

Many problems, for example the maze experiments, display regularities in the fitness landscape, and primarily for this reason we seek more complex search mechanisms that can take advantage of these regularities.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Length of the program.</td>
</tr>
<tr>
<td>$PROVISION_{MAX}$</td>
<td>Maximum cumulative reward allowed.</td>
</tr>
</tbody>
</table>

With the described above experimental setup, we have observed the following features:

- Cells create connections by executing neighbours’ programs and use the self-assembly process, i.e. autonomous generation of structural organisation (see Figure 9.12). Cells group together, forming self-maintainable ensembles (more stable as they receive more rewards) to solve more complex tasks. This confirms our preliminary results from Section 9.3 and links the experimental work with research of autopoietic hypercycles.

- The system is self-organising, because its structure appears without explicit predefined design or explicit involvement from the outside environment. Cells must autonomously find appropriate connections. Note that autonomy here means purely statistical autonomy. Especially in exclusively random search process, even the process of finding connections can be considered
an emergent property of the system. We use the terms *autonomy* and *learning* in a metaphorical sense, as these are both macroscopic properties and results of microscopic rules and interactions.

- The environment representing the tasks to be solved is dynamic and the system exhibits self-adaptable properties. Cells keep track of environmental variation: if established programs are no longer successful, i.e. no rewards are obtained, they are (dynamically) replaced by new ones. Cells thus react to the dynamic environment, and our cellular system is self-adaptable.

- Thanks to its high redundancy and massive decentralisation, our cellular system exhibits flexibility and robustness – it can work with many independent task hierarchies.

- There is almost no representational bias on the individual cell level, due to the base EVM capabilities, i.e. full computational search space. The representation does not rely on some expert knowledge or any similar bias mechanism *ab initio*. The main objective here is to investigate raw properties of the search mechanism and how the cells will build the bias on their own, autonomously.

- Our system automatically adapts its inductive biases, i.e. meta-learns, by taking advantage of the correlations between tasks (reusing previous solutions or parts of them).

- Shorter solutions are more likely to appear, thus efficient and compact information coding dominates. This also causes the search to get stuck on sub-optimal solutions that are much shorter and compact, but exhibit better generalisation properties.

- Some emergent properties have been observed, for example, the way knowledge is diffused thanks to parasites and parasitic/symbiotic interactions.

- Our system exhibits several features of artificial life systems: cells struggle to survive, and exhibit competition, collaboration, symbiosis (mutualism and parasitism), exaptation, locality, and catastrophes.

- Due to the high dependency on local interactions, some cells may fail to access the useful building blocks appearing in further areas of the grid. This accessibility issue is of major concern. In particular, it affects the scalability of the presented above model.

- The environmental density may be difficult to balance between a too sparse (no useful neighbour) and a too dense/saturated (no room left for new solutions) web.

Based on the obtained results from the described phase of experimentation, we have decided to continue the research on this model, in particular, conduct more extensive experiments, and quantitative analysis of the impact of some parameters. To address the drawbacks of locality and density
of the environment, we decided to prepare a modified iteration of the EVMI architecture, with a network-like model, which provides much more robust and flexible connectivity model.

This particular model provided much of the insights into the workings of the EVMI, exhibited most of the expected emergent properties and proved promising results in aspects of: self-organisation, exaptation, symbiotic linking, and self-assembly. Not all the properties and behaviour is well and completely understood yet. There is space for further explorations and study.

9.9 Stochastic Search mechanism

9.9.1 Description

For the stochastic search we assumed that the number of instructions per program (and also the number of programs per cell if there is more than one) is limited to a fixed number\(^7\). The basic idea is to assign and dynamically manipulate a probability distribution for each instruction. For each instruction of the cell’s program \(P\), there is a probability distribution over all of its possible values (Figure 9.14). One can visualise \(P\) as a space of all possible programs \(p\) with given probability distributions, from which a particular \(p\) is being generated.

![Diagram of probability distributions for instructions]

Figure 9.14: Every instruction of the cell’s program \(P\) contains a probability distribution over its possible values.

The value \(p\) for the cell’s program \(P\) is selected as follows; for every instruction of a program \(p\), randomly pick a value according to its probability distribution. Depending on the success of \(p\), probabilities of the picked values will be slightly increased or decreased dynamically, following a predefined probability change and the value of the reward achieved by \(p\) (see below).

\(^7\)There is no intrinsic limitation to using a flexible variable length data-structures instead of the fixed-length ones. It is mostly due to computational resources and performance considerations that we have used the fixed-length programs.
9.9.2 Management of probabilities

Every time a reward $r$ is gained by a program $p$, we increase the probabilities of $p$’s instructions. Every instruction $i$ in the cell’s program $P$ has its probability distribution modified:

$$P(p_i) \rightarrow P(p_i) + \alpha r,$$

where $p_i$ was the selected value for the $i$-th instruction of $p$. To maintain the sum of the probabilities equal to 1, all other values of the instruction set $I$ have their probability proportionally decreased:

$$P(j) \rightarrow P(j) - \frac{\alpha r}{|I| - 1}, \forall j \neq p_i.$$

If no reward has been obtained, we slightly decrease the probabilities of $p$’s instructions. For every instruction in the machine,

$$P(p_i) \rightarrow P(p_i) - \beta,$$

where $p_i$ was the selected value for the $i$-th instruction of $p$. All other values have their probability proportionally increased:

$$P(j) \rightarrow P(j) + \frac{\beta}{|I| - 1}, \forall j \neq p_i.$$

Two constraints have been added to keep the entire search space accessible for the search process of a cell. All probabilities must stay between a lower bound $P_{min}$ and an upper bound $P_{max}$:

$$P_{min} \leq P(i) \leq P_{max}, \forall i.$$

This condition is necessary to maintain a proper balance between exploration and exploitation. Without these boundaries, the search would only exploit the first sub-optimum found, by decreasing all presumably irrelevant instructions to zero probability of being selected. This excessive pressure for exploitation would contribute to serious over-fitting.

9.9.3 Observations

**Correlation between instructions.** The primary drawback of the stochastic approach (and to a certain extent, GAs as well) is that it does not take into account correlations between individual instructions (instruction sequences). In all our testing scenarios a program’s quality highly depends on all its instructions taken together, in an appropriate sequence. Changing one of them often disrupts the entire program and penalises the other instructions, even if these are likely to be beneficial. Storing conditional probabilities of instruction sequences might be a potential solution to that problem. This is why a mechanism for storing successful subprograms’ patterns of any length in a probability tree has been proposed, and it is presented in the next section.
Figure 9.15: Evolution of probability distribution for an instruction: typical examples. Top figure: No reward has been gained yet. The system is still in an exploration phase. The different values tried have always been unsuccessful (thus punished). Bottom figure: the mechanism found a good value for that instruction. This value gets rewarded, thus increasing its probability, and likelihood of being picked more often, rewarded more often, etc. This kind of autocatalytic reaction increases its probability to the maximum. The speed at which the probability increases can be adjusted with the parameter $\alpha$. 
Figure 9.16: The risk with the stochastic method is that if $\alpha$ is too small (or $\beta$ too big), no value has enough time to take over. Indeed, once a good value gets rewarded (peak on the figure), it will be picked more often. But to be rewarded again, good values need to be picked for the other instructions of the program, too. And this requires a couple of attempts. At every unsuccessful attempt, the probability of the good instruction is decreased. If it wasn’t high enough, it will progressively be forgotten. Here, for instance, good values have been forgotten three times. On the other hand, if $\alpha$ is too big, the program will converge too quickly towards the first suboptimum it comes across.

Colonisation of programs. In the case of a machine (several programs) per cell, solutions tend to colonise all the programs of $M$. For instance, if the environment provides 4 problems, and $M$ has got space for 10 programs (note, in this case, we used fixed-length data-structures), every program of $M$ will converge towards the same solution. The reason is that programs take considerable amount of time to converge, thus once the first 4 programs of $M$ have “stored” the 4 solutions, they will take all the rewards and leave nothing for the other programs.

Freezing mechanism. Since instructions are randomly selected according to their probability distributions, there is always a probability of not selecting an important instruction $i$ for $p$. To be sure that some of the critical instructions $i$ are in fact in the generated program $p$, the search mechanism tends to increase its probability in consecutive placeholders as well. Therefore, the search wound up confined to local optima, such as 

$[\text{right down right down down right down}]$ \((S3)\),

for maze experiments 1 and 2. This was especially disappointing, because during the evolutionary process, the search mechanism could find some good solutions (up to 66 reward units in the second experiment) but would fail to remember them. The search process can be enhanced by introducing a freezing mechanism that will progressively freeze values in the cell’s program’s instructions (see Figure 9.17).

To guarantee adaptability, the freezing mechanism must have a reverse (unfreeze) mechanism in case of changes in the environment. We unfreeze the last frozen instruction of $P$ if the selected
(1) Assign $n \leftarrow 1$ (start from the first instruction).

(2) If the probability of the $n$-th instruction has been almost maximal (within predefined threshold $\theta_1$) for a given number of iterations (another predefined constant $\theta_2$), set the probability to 1 and don’t modify it anymore (freeze it).

(3) Reset all the probabilities of the tail of the program, i.e. for all the instructions $n + 1$ and above.

(4) Set $n \leftarrow n + 1$ and return to step 2 until the entire program has been frozen.

Figure 9.17: The freezing mechanism.

The freezing mechanism has proved to be highly effective in many different problems we have tried, and it has always outperformed the stochastic search without freezing. In experiments 1 and 2 the search converges nicely to solutions containing a loop, either with 2 or 4 agent movements instructions inside it (the more movement instructions, the higher the total reward). For instance, the second experiment produced these solutions:

- `[down right right left const_2 goto]` (50 reward units)
- `[down right right down left up const_2 goto]` (66 reward units)

The first program moves the agent: down, right, right, then left. After that sequence, a value of 2 is placed on top of the stack (instruction `const_2`, and instruction `goto` moves back to the third
instruction in the program (right). The sequence of agent movements: right, left is executed indefinitely in the loop.

**Long programs.** Even with the help of the freezing mechanism it became difficult to build up longer programs, especially when a shorter (yet less rewarded) one was relatively easy to find. The third experiment demonstrates this. Instead of finding the better 12-instructions-long program, all of our search methods attempted to find a pattern with the use of a loop, e.g.:

\[
\text{[right down right down down const}_0\text{ goto]}(7 \text{ reward units})
\]

Since this program is shorter, it is more likely to be discovered by the search process itself. It gets smaller rewards than the best solution, but it gets them more often, and for this reason it dominates. Exploitation takes over the exploration process easily. The issue of the scaling of rewards in the function of program length becomes apparent here. There is however no dynamic mechanism for this sort of scaling available at the moment, due to a complete lack of information of the fitness landscape. Usually, the length of the program versus the quality of the rewards will be controlled by external parameters, such as the available computational resources to tackle a given problem, and will have no direct correlation with the actual original fitness landscape of an arbitrary unconstrained problem.

To some extent these issues have been tackled by the use of the stochastic tree method, which will be discussed in the next section.

For the case of several programs per cell, just as for random search, the same mechanism is applied, in parallel, to all programs of the cell.

### 9.10 Tree mechanism

#### 9.10.1 Objectives

As explained previously in Section 9.9, the main drawback of the stochastic method is that it does not take correlations between the program’s instructions into account. Let us consider an example to highlight this problem.

Suppose the solution is the program \text{dec const}_2\text{ add swap}. To remember it, the stochastic method will increment the probabilities of \text{values dec, const}_2, \text{add, and swap for instructions 1, 2, 3, and 4 respectively. However, other sequences containing add at the third position, as dup dup add const}_0, behave completely differently and are punished. They decrement the probability of value add for the third instruction and contribute to removing the information about the solution. Therefore, even if, by chance, a good solution is found once, it is quickly forgotten due to the influence of other, not-useful programs. The longer the solution is, the more such disruptive sequences there are in the search space that disrupt the bias of good instructions. The longer the solution, the less likely it becomes that the stochastic method will find the solution at all.
Table 9.4: Parameters for the stochastic search specialisation mechanism.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Length of the program.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>How much probabilities of successful programs are increased.</td>
</tr>
<tr>
<td>$\beta$</td>
<td>How much probabilities of unsuccessful programs are decreased.</td>
</tr>
<tr>
<td>$P_{\text{min}}$</td>
<td>Minimal probability.</td>
</tr>
<tr>
<td>$P_{\text{max}}$</td>
<td>Maximal probability.</td>
</tr>
</tbody>
</table>

Additional parameters for the freezing mechanism:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta_1$</td>
<td>Threshold to bypass to be frozen.</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>Number of iterations needed (if above $\theta_1$) to be frozen.</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td>Number of iterations needed (if no reward) to be unfrozen.</td>
</tr>
</tbody>
</table>

To address these issues, we have proposed a modification to a simple bias-optimal search based on the notion of sequence conditional probabilities.

### 9.10.2 Conditional probabilities

This problem of forgetting good solutions because of the influence of useless solutions can be reduced by introducing conditional probabilities. Instead of incrementing $P(\text{add})$ for only the third instruction, we increment $P(\text{add}|\text{const}_2)$, for a sequence of 2 instructions. The algorithm can store conditional probabilities up to arbitrary length, although from experimental studies it appears that sequences longer than a few instructions are the most effective. The cost of storing all the conditional probabilities increases exponentially with the number of previous instructions taken into account. For more than 3 previous instructions, it quickly becomes unmaintainable. Also, the problem will be reduced, but not removed or fixed. Indeed, if the solution is 10 instructions long, using relative probabilities of order 3 may not be enough. Finally, most of the memory in such a model would be wasted with the storage of probabilities that are never used. As attractive as the full-featured conditional probability model appears to be, it becomes apparent how difficult and inefficient it would be in practice, if implemented in a naive way. To make use of program sequences and conditional probabilities a better model needs to be proposed.
9.10.3 Storing patterns

In the ideal case, the whole sequence of instructions should be stored. The mechanism must only remember that \texttt{dec const 2 add swap} was a good program. No matter that \texttt{dup dup add const 0}, or \texttt{dec add add mul} are bad. Ideally the system should just store probabilities of instructions of successful programs. That can be achieved by using a tree-like data structure (Figure 9.19). Every time a reward is gained, the probability of the successful sequence of instructions within the tree is increased. We called this model a tree model, or tree-based bias.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{tree.png}
\caption{A tree-like data structure is used to store the probabilities of the successful subsequences (patterns). Nodes for instructions’ values that don’t appear in the tree have all equal probability (their value is computed such that the sum of all probabilities is 1). Probability 0.6 for the orange \texttt{dec} node means that $P(\texttt{dec}|\texttt{add}) = 0.6$. In other words, if the first instruction is \texttt{add}, the probability of it being followed by \texttt{dec} is 0.6.}
\end{figure}

9.10.4 Reward mechanism

There is a subtle question of what should be rewarded. In other words, what part of the program was responsible for its success. Some instructions (introns) may actually do nothing and could potentially be left unrewarded. But, is it possible to identify the introns and adjust the probability distribution biases accordingly? Let us illustrate it by an example.

Suppose the shortest solution is the program \texttt{dec const 2 add swap}. Sequences with introns, like \texttt{dec nop const 2 add swap}, or \texttt{dec const 2 nop nop add nop swap} behave exactly the same and thus receive the same amount of rewards. If we increase the probabilities of the whole sequence, the tree will never converge towards a unique solution. In some way, we have
to extract what all these rewarded programs have in common. Our proposition is to reward all the possible subsequences, even the ones that skip instructions. This number is big \((2^l - 1)\) for a program of length \(l\), but ensures that the common pattern present in all solutions will be effectively found.

If a program \(p\) gains a reward \(r\), all the possible subsequences of \(p\) have their probability increased by \(\alpha r\). An example is presented in Table 9.5 and in Figure 9.20.

**Figure 9.20:** Simple example of the way the rewards mechanism works. The previous tree was the one of Figure 9.19. \(r = 10\) rewards units has been obtained with the program `dec mul`. Suppose \(\alpha = 10^{-2}\). Thus, we have to increase by \(\alpha r\) the probabilities \(P(\text{dec})\), \(P(\text{mul})\), and \(P(\text{mul}|\text{dec})\). Suppose there are 10 possible values (\(|I| = 10\)). To keep the probabilities coherent, the other probabilities with the same parent node must be decreased by \(\alpha r/(|I| - 1) \approx 10^{-2}\).

Table 9.5: All possible subsequences of \(p = \text{dec const}_2 \text{ add swap}\). Notice that we allow instructions to be skipped, but the linear order must be maintained.

<table>
<thead>
<tr>
<th>dec const(_2) add swap</th>
<th>dec add swap</th>
<th>const(_2) add swap</th>
<th>add swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec const(_2) add</td>
<td>dec add</td>
<td>const(_2) add</td>
<td>add</td>
</tr>
<tr>
<td>dec const(_2) swap</td>
<td>dec swap</td>
<td>const(_2) swap</td>
<td>swap</td>
</tr>
<tr>
<td>dec const(_2)</td>
<td>dec</td>
<td>const(_2)</td>
<td></td>
</tr>
</tbody>
</table>

Another relevant question is when to reward. The answer to this question depends on the problem at hand, and has been investigated by other life-long learning algorithms, e.g. (Schmidhuber, 1995). For these experiments we have decided to pass the reward immediately every time the program gets a positive reinforcement. That allows us to use a maximum amount of information about what part of
the program was responsible for the rewards. Indeed, if for instance an intron is added at the end of a successful program, sequences with this intron are not rewarded, because the reinforcement feedback was present strictly after the reward was obtained.

9.10.5 Selecting a program

A program’s instructions are selected in an incremental way. To add an instruction, we look for the longest branch that fits to the current program. According to the probabilities of the children of that branch, we pick randomly the next instruction.

For example, to add an instruction to \texttt{sub const3 dec}, we check if there is a branch \texttt{sub const3 dec} in the tree. If such a branch does not exist, we check for \texttt{const3 dec}, and then for \texttt{dec}. If nothing is found, we take the root node and a single instruction probability to generate a new instruction. An example for selecting the program is presented in Figure 9.21. Let us explain how it works in that particular case.

\textbf{Step 1.} The first instruction is selected according to the probabilities of the first layer of the tree (the root of the tree). Notice that all the possible values not stored in the tree have an equal minimal probability (such that the sum equals 1).

\textbf{Step 2.} Since \texttt{add} has been selected for the first instruction of \(p\), we look for the conditional probabilities \(P(i|\texttt{add})\) to choose the value for the second instruction, i.e. for the probabilities of having a pattern of the two instructions \texttt{add i}.

\textbf{Step 3.} Since \texttt{dec} has been selected for the second instruction of \(p\), we look for the conditional probabilities \(P(i|\texttt{add dec})\) to choose the value for the third instruction, i.e. for the probabilities of having a pattern of the three instructions \texttt{add dec i}.

\textbf{Step 4.} Since \texttt{nop} has been selected for the third instruction of \(p\), we look for the conditional probabilities \(P(i|\texttt{add dec nop})\). There is no information in the tree about this conditional probability. So we look for the \(P(i|\texttt{dec nop})\). Since there is still no information, we look for \(P(i|\texttt{nop})\). We select therefore the forth instruction according to the probabilities of having a pattern \texttt{nop i}.

9.10.6 Additional mechanisms

\textbf{Limited size.} Obviously, with a large number of subsequences to store in the tree, the tree size will quickly explode in size and become unmanageable even for moderately long programs. We decided to constrain the number of nodes to a certain upper limit. Once that limit is reached, in order to provide space for new entries, we adapted the following procedure:

- If the tree is full and we want to add a node with a value\(^8\) \(v\), find the node in the tree with the

\(^8\)The value of a node \(n\) is the probability of the branch starting with the root and finishing with \(n\). In Figure 9.19 for example, the value for the node \texttt{mul} of the third layer is \(0.2 \ast 0.8 \ast 0.7 = 0.112\).
Figure 9.21: Program selection example. In red: the longest branch that fits to the last chosen instructions of \( p \). In blue: the children nodes of that branch. These blue nodes are the information used to pick the value for the next instruction.
smallest value $v_{min}$. If $v_{min} < v$, remove that node and insert the new node in the tree.

- At every time step, slightly decrease all the values in the tree (a forgetting factor).

It is important to notice that even if a node is removed from the tree, its information does not disappear completely. Since a parent value is the sum of all its children, the contribution of the deleted child still remains in its parent’s value.

**Freezing.** After the first set of experiments, it was clear that short programs were advantaged, due to the nature of a reward system promoting all the nodes close to the root of the tree. In the maze experiment number 3, for instance, a loop (with reward value 7$) was always preferred to the best solution (12$). To demonstrate the point, consider these two solutions $p_A$ and $p_B$ (Figure 9.22). $p_A$ is 4 instructions long, and $p_B$ only 2. Even if $p_A$ gets bigger rewards, it won’t take over, because $p_B$ has a higher probability\(^{9}\) The reason is that the longer the program, the less likely it is to be generated in the random fashion.

![Figure 9.22: Without the freezing mechanism, a long sequence is less likely to be picked, preventing the mechanism from converging towards long solutions.](image)

To remove this undesirable property and promote longer solutions, additional freezing mechanism has been added. Similarly to the freezing mechanism of stochastic search (Section 9.9), it freezes a probability to 1 when a value of a node has been above a certain threshold for a given number of iterations. For the sake of adaptability, there must exist a reverse process to unfreeze nodes when they are no longer successful. In other words, starting from the leaves, when a probability is above a certain threshold for a certain amount of time, it will set it to 1 – this process is referred as freezing.

\(^{9}\)Of course, if $p_A$ gets substantially more rewards than $p_B$, it will take over, eventually. We want to highlight that short programs are advantaged when no proper reward scaling is present. Long programs need to be substantially better to influence the search, otherwise short, compact and general solutions with the use of loops and internal substructures will always win out over long and elaborate solutions.
In the case of several programs per cell, the same mechanism is applied, in parallel, to all programs of the cell. In other words, there is one tree per program.

**Implementation.** As with the stochastic method, all values in the tree are integers, thus accelerating all the computations of our implementation. The program selection mechanism is implemented in a fast parallel way, but to keep the tree size below the limit, we have to maintain pointers on the lowest values (the firsts to be removed) in the tree. That considerably slows down the process.

Notice that this mechanism enables us to select the amount of memory used (size of the tree). Also, the memory is used in an efficient way, as the nodes stored are only successful ones.

Table 9.6: Parameters for the tree-based specialisation mechanism.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>How much probabilities of successful patterns are increased.</td>
</tr>
<tr>
<td>( \beta ) (Limited size)</td>
<td>How much probabilities in the tree are decreased at every time step.</td>
</tr>
<tr>
<td>Tree size</td>
<td>Maximum number of nodes for the tree.</td>
</tr>
</tbody>
</table>

The tree-based stochastic search was the best performing learning algorithm used within the context of the currently implemented EVMI. It can be successfully applied to toy-problems, and we have tested it on various arithmetical and maze experiments. The attached CDROM contains the environment and collected results of various runs together with various films and snapshots of more interesting results obtained to date.

### 9.11 Summary of experiments

A number of experimental studies have been conducted. We have used various EVMI configurations and analysed various mechanisms, stressing the self-organisation and adaptability of the search process itself. The most notable results and discoveries of the experimental studies are:

- **cell specialisation:** cells automatically specialise to solve various distinct tasks, based on their own trajectories of development, environmental pressures (reward structure) and developmental costs (cells that are faster in solving a given task establish themselves earlier in the emerging structure).

- **knowledge diffusion:** cells solving a given task must spread their capabilities through a complex web of parasitic and symbiotic relationships. Further studies in the area of tags and refer-
rals are needed to augment the existing low-level spontaneous mechanisms with more abstract communication based abstractions.

- self-assembly of the EVMA cells: cells spontaneously organise themselves into larger structures to benefit collectively from the reward schemes and positive feedback loop.

- automatic complex task decomposition: although limited in scope, the EVMI tested so far exhibited the ability to decompose complex computational tasks into a set of simpler tasks, which when assembled are capable of solving more complex tasks. The ability to solve complex computational tasks by a set of highly specialised simple and autonomous computational cells has been demonstrated. For practical deployments more exhaustive studies into the influence of cell topology and internal search biases will be necessary. Nevertheless the principal phenomena have been demonstrated empirically.

With regular grid experiments the video sequences proved to be an excellent visualisation and study tool. However, for the generic EVM Universe experiments, with no-regular topology and with cells changing the local neighbour relations frequently, the basic movie technique could not be directly applied anymore. Studies in network-like (or graph-like) EVMA models continue to be challenging. At the time of this writing, the proper methodology has not yet been established. We continue to work on better models and simulation techniques. Our current approach is to extract a single interesting aspect of the running simulation and analyse that single aspect in isolation. This however, has its obvious drawbacks, i.e. the non-obvious relationships between certain aspects of the system are not easily discovered or demonstrated. Unfortunately, search spaces based on computing languages, and the EVMA in particular, are inherently complex and discontinuous. Such investigations are extremely time consuming, complex and tedious. We envision that the film-based technique will play a major role in this study area, but will need to be extended into multiple dimensions and improved. This is indicated in the next chapter as an area for further work.
Chapter 10

Summary
10.1 Summary and Comments

In this thesis, the Evolvable Virtual Machine abstract architecture for hierarchically organised evolvable and adaptable virtual machines has been presented. The EVMA builds on Turing-machine, stack-based traditional models of computation, and extends it through the notion of an autonomous and asynchronously interacting collection of computational cells. It can be used as a generic computational modelling framework. However, the primary intention and usage of EVMA is to provide a comprehensive and effective unified platform to integrate life-long reinforcement learning and traditional computing paradigm together. The work combines several, quite diverse research areas, ranging from system sciences, cybernetics, computational and biological evolution, theories of hypercycles and autopoiesis, through reinforcement and meta-learning. This study that extends over multiple disciplines provides a synthesis based on various models of evolution and adaptation, which are generally labeled as processes of life. The main stress of the work is on life-long reinforcement learning in multi-task environments through continuous refinement and reification of the computational substrate itself. When implemented in hardware in massively multi-core systems it offers the ability to on-the-fly monitoring and adaptation of computational architecture to take advantage of the regularities and patterns that are present in the computational processes.

10.2 The EVM as a machine learning system

Advances in machine learning and evolutionary systems have stressed the benefits of learning how to solve several, possibly related, tasks through reuse of common patterns and behaviours. This is referred as multi-task learning. The ability to learn how to learn is referred to as meta-learning. The EVMA provides an adaptable abstract software architecture, together with a prototype implementation (EVMI) for multi-task and meta-learning. The EVMA takes its inspiration from contemporary biological evolutionary theories. The EVMI is built on top of a stack-based cellular system architecture that exhibits certain patterns of behaviour observed in molecular and evolutionary biology. It has been shown experimentally that specialisation, together with symbiosis and exaptation, are useful (often necessary) elements for any adaptive system to reach high complexity levels.

The EVMI as a cellular systems technological platform consists of many simple, locally interacting components, called cells. Because of high redundancy, decentralisation, and a massively parallel asynchronous architecture, an EVMI provides a highly efficient, adaptive, and robust learning infrastructure.

10.3 The EVM as an artificial life simulator

An EVM’s features support and enable exploration into the realms of artificial life. Cells compete for environmental resources, but at the same time, they also co-operate by symbiotic interactions.
Some cells engage into parasitic relationships by using other cells’ programs to calculate the solution. Multiple cells sometimes engage in mutually-benefiting relationships to solve tasks together. Experiments, together with qualitative analysis, have been performed to compare the different specialisation mechanisms and to observe the global response to different environmental pressures. They also bring up interesting issues about locality (accessibility of useful primitives) and scalability.

Bio-inspired computing machines can be partitioned along three axes, depending on what level of organisation of life on Earth they take their inspiration from (Sipper, Sanchez, Mange, Tomassini, Pérez-Uribe, and Stauffer, 1997). The first level, phylogeny (P), concerns the temporal evolution of a genetic program. The second level, ontogeny (O), is the developmental process of a multicellular organism from its own genetic material. Epigenesis (E) is the third level, and loosely corresponds to lifetime learning through environmental interaction. Together, these three axes define the POE model (Sipper, Sanchez, Mange, Tomassini, Pérez-Uribe, and Stauffer, 1998).

The GA-based cell specialisation mechanism belongs to the phylogenetic process, acting at the cellular level. The overall emergent and self-organising properties of the EVM system, through the interactions with the environment, belong to the epigenesis level.

Ontogeny, the developmental process of the multicellular organism, can be viewed as a two-fold process. First, as the composite connected program based on a collection of interacting cells, or, as complex intra-dependencies between the machines on the single cell level. These aspects however have not been investigated in this work, and are left for future studies. Many cellular systems are based on the concept of ontogeny (especially in multicellular automata) (von Neumann and Burks, 1966; Langton, 1984; Marchal, Piguet, Mange, Stauffer, and Durand, 1994; Mange, Stauffer, and Tempesti, 1998), and it is desirable to investigate how the growth of cell assemblies can be controlled and manipulated more efficiently. It could lead to further studies of biological-like properties, such as replication or regeneration (self-repair).

10.4 The EVM as a cellular system

Cellular systems consist of many simple, locally interacting components, called cells. The main idea is to use the high redundancy and massive decentralisation. Cellular systems have proved to be highly efficient, adaptive, and robust (Sipper, 1999; Capcarrère, 2002). A cellular system on the macro-level exhibits properties that are not apparently obvious from the micro-level. These phenomena are usually referred to as emergent properties. The notion of emergence have been studied in diverse fields of computer science, e.g. Cellular Automata (Wolfram, 1984), Membrane Computing (Teuscher, 2004), or Ants Algorithms (Bonabeau et al., 2000).

Cellular Computing is a general paradigm that describes computation as a massively parallel decentralized process whose result is usually taken to be the state of the system as a whole (Capcarrère, 2002). It encompasses the classical model of cellular automata. It models asynchronous, non-deterministic or unstructured variants and morphogenetic systems, as well. Membrane comput-
ing could be considered to be part of the more general idea of cellular computing. The main ideas of interest to such systems are usually associated with the high redundancy and decentralized nature of the systems, and that the emergent nature of the computation lead to robust and adaptive process models.

Flexibility and robustness are primary advantages of cellular systems. Because of the absence of global, central control, decentralised cellular systems are not prone to major failure when any single element fails. All components operate only with interactions, and the whole system does not rely on any individual in particular; thus none is critical for the overall performance. The large number of components in a cellular system allows individual cells to be relatively simple, without sacrificing the computational power of the overall system. Also, all these simple components are easily replaceable.

It seems that although some systems are inherently simple, and others, for example a human society (as a cellular system), or the world wide web with inter-connected mobile devices, have inherently more complexity in the individual cell. Although in majority of experiments each cell contains a single program, it is possible for a cell to contain more than one program, or even, to be composed of a little EVM Universe itself. The implications of such an organisation are beyond the scope of this work, and represent grounds for promising research activities in the future.

In some configurations, a cellular system interacts with an external environment, for example cells try to solve certain externally provided tasks. If they are successful, they receive rewards (food) from this environment. Our EVM architecture can be treated as such an cellular system, where every cell asynchronously evolves a program in order to specialise in solving one of the tasks. We introduced and compared five different cell specialisation mechanisms to achieve this process. They operate at the low, cellular level, but influence the global, macroscopic behaviour of the whole system.

In cellular systems, the important question is how complex are the individual cells, at the bottom cellular level. It has been commonly acknowledged (Sipper, 1999) that individual cells should preferably be simple (like the few rules and states of cellular automata). Conceptually, however, nothing really conflicts with the idea of more elaborated cells or even intelligent agents. The main idea here is that the EVM can in principle model interactions and computations on many different levels, including the computational fabric as well as the applications running on that fabric. The inherent side-effect is however extremely large computational costs associated with increased complexity of the lowest level.

A simple case, showing how cells can self-assemble to solve tasks of increasing complexity has been described in Chapter 9.

10.5 Computational limitations

We note that all of the self-learning and self-adaptable algorithms (such as ADATE, EIRA, and the EVM architecture) are theoretically sound, and represent models close to our current understanding of biological evolutionary processing. However, they all suffer the limitations related to computa-
tional complexity. In other words, the computational resources required to perform the full reification and self-learning are prohibitively large for complex tasks, for which less adaptive and more tailored and customised evolutionary techniques are better-suited and work more efficiently. From the theoretical, architectural and evolutionary perspectives, however, it is not crucial to solve actual problems, but rather, to investigate the properties and dynamics of the evolutionary processes, so we can understand the processes involved in more detail. Computational limitations will be (at least partially) addressed by the advances in computational hardware systems based on massively multi-core CPUs. The EVMA may help in influencing the next generation virtualisation layers in modern ISAs.

10.6 Research issues and future work

We envision that the EVMA will help other researchers to conduct experiments and further progress the studies of bio-inspired computational architectures, and instruction set architectures in general. Others should find the EVMA abstractions of computation, evolution and the process of life useful and applicable to their own respective disciplines. In particular, the EVMA might provide useful insights for the fields of evolutionary biology, genetics, artificial life, computer science and philosophy. The EVMA opens up new research and development pathways not only within EC and ALife research, but also within the general notions of computation and computational architectures.

Below we list some of the open research areas related to EVMA:

• **Reusing existing code.** To facilitate the reuse of existing code written by human programmers and/or discovered by other code generators, there is a need for automated translation tools that would compile/translate existing source/binaries into the EVMI assembly language. Once appropriate compilers and translators are provided, for example for Intel x86 assembly and Java, it will become possible to implement all of the core EVMI libraries in the EVMI assembly, bootstrapping the future EVMI implementations within the EVMA framework itself.

• **Evolutionary computation.** The EVMA generalises and encompasses many of the existing models and techniques used within EC. Thus, many existing EC models can be seen as special cases of the EVMA, and further studies can be conducted to formally express various EC methods in a unified EVMA framework.

• The computational aspects of autopoiesis and hypercycles need to be explored in more detail. The mechanisms and self-organising principles that are developed for computational autopoiesis and hypercycles may provide insights into biological organisation and detailed dynamics of evolutionary processes.

• **Evolutionary biology.** One of the possible applications of the EVMA is its use as a formal model for evolutionary theories on different levels of abstraction. The formalism presented in this thesis can be used as an operational definition of a living system. Further detailed
formalisation of the framework may be necessary, and concrete applications to evolutionary biology would need to be established.

- The use of EVMA abstractions in discussing complex dynamical systems and evolutionary dynamics in particular has been provided throughout the thesis, but can nevertheless be extended to other physical and bio-chemical studies, especially those that use primarily computational modelling techniques.

- **General-purpose computing architecture.** With the rapid advance of multi-core and virtualisation technologies, new computational paradigms are needed to help programmers utilise efficiently and effectively the growing number of independent computational resources. The EVMA provides the means to automate many of tedious runtime tuning and optimisation tasks with the built-in ability to restructure the computational architecture. EVMA can specialise the virtual machine, and take advantage of the regularities that may occur between various computational tasks being executed together on a given platform.

- Computational complexity of meta-learning algorithms and bias-optimal search needs to be explored in further in the context of the EVMA and specific problem domains. It is possible that computational architectures and grid computing will become sufficiently powerful that enumerations will dominate the search technologies in program spaces. In that case EVMA-like architectures may well become dominant for general purpose computing and runtime code optimisation. Formal bounds on computational complexity of various meta-learning and bias-optimal search techniques would provide additional information for optimisation. This can be done for example on the operating system level, or on virtualised domains running alongside primary operating system domains. Preliminary research in this area is conducted on Sun Microsystems OpenSparc T2 CPU family with the Netra Data Plane System (NDPS) logical domains virtualisation.

- The notion and possibility of hypercomputing based on a massively asynchronous computational model sketched in this thesis has get to be fully verified and extensive further studies and investigations into the nature of massively asynchronous systems based on interactive computation may provide further insights and enhance our technological/computational capabilities.

## 10.7 Complex interaction networks

### 10.7.1 The EVMA and the Internet

When taken as a whole, the Internet cannot be modelled as a single user and single computer von Neumann model (or Turing machine for any practical analysis). The von Neumann model is not adequate to the current complex and interlinked computing Internet. The Internet itself is simultaneously
computational, social and economic system, with data and information being generated from within, taken out and put in from the environment.

The previous advances in the IT technologies were based on abstracting from hardware. The abstraction layer has been achieved through various software virtualisation solutions and virtual machines. The new shift in is based on abstracting from software itself. All computations and all resources are often abstracted out as services, with the actual software components being virtualised. In a world of services it is the service itself that counts – not the hardware or specific software components. It is the computational transformation that a given service provide that is important. Efficient transformation can be done by human programmers, or can be assisted by the automated tools that take advantage of speculative execution for example. The EVMA can model and assist in changing the existing human-driven software generation, into semi- or fully-automated process.

With the growth of the Internet and the ubiquitous service orientation, solutions need to scale up to open and dynamic environments of billions of services (or computational cells)\(^1\). With such large scale systems, such as many portals and data warehouse applications, it is the human aspects of the system that become the bottleneck. It is the human programming in the loop that prohibits a given solution to scale well. Automation of programming and adaptation of software are important aspects of the dynamically changing IT landscape. The EVMA architecture by addressing those complex issues indicates automated ways of software development with limited amount of human intervention. The EVMA can be used to model or even host service-oriented computational architectures, where some (or most) of the actual programming and software adaptation is done automatically. Our current results show that this can be achieved successfully for small computational tasks. Further work is needed to investigate how complex computational tasks can be facilitated by a given EVMI. As indicated earlier, hardware accelerated EVMI would probably help in tackling larger and more complex problems.

### 10.7.2 Application to genetic networks

Inspired by the theory of biological evolution the EVMA architecture can be used to model and discuss in computational terms many new emerging hypothesis from the evolutionary biology. The most interesting open research areas are, particularly, in the interactions between genes proteins DNA and RNA – the complex hierarchical genetic networks with multiple feedback loops. Garrett Odell stated: “the next step [in genetic research] will be to figure out the interactions within networks of networks”(Meir, Munro, Odell, and von Dassow, 2002). This is where the EVMA can help in modelling, especially for protein-protein interactions – identification of molecular machines and signal transduction cascades. Often a set of mutually cooperating genetic transcription elements can bind to the control regions of many genes to enhance or inhibit transcription, or signaling an regulatory

\(^1\)There is currently more than 1 billion Internet users, more than 30 billion web pages, and over 18000 web services, doubled in the year 2006 (see http://www.seekda.com). The number of web services is going to grow exponentially.
pathways and protein complexes. These consist of genes proteins and small molecules wired together in a complex network of inter-molecular interactions – something the EVMA can be well suited to model in computational settings.

10.7.3 Reflection

Any physical computational process is unable to completely self-reflect on itself, due to the inherent infinite regress of reflection. In other words, computational process is unable to reflect on itself to the level so as to achieve a full reflection of itself – it would require more computing capabilities that a given process contains. Thus, the only way for computational process to reflect on its own computation is to use an outside environment as a mirror, or source of additional computing resources necessary for the reflection process. An EVM cell needs other cell, the environment, as a mirror to achieve the arbitrary level of reflection of itself. The reflection of self needs to go through a feedback loop from the outside. Again, the EVMA is designed to be well suited to model such computational reflective architectures. Through explicit modelling of computational dependencies and the ability to express hierarchies, the EVMA is well suited for studies on computational and self-reflective software systems. Further work is needed to provide in-depth analysis of how much complex software systems can take the advantage of this type of reflective processes.
Appendix A

Notes on information theory

A.1 Measures

Information is a measure based on a selection from a set of available choices. Algorithmic information is a uniform measurement of encoding of information relative to the given computing machine (virtual machine).

The amount of information is based on the ability to make a correct selection from a given set. Let us consider a unique code \( k \subset X \times Y \), i.e. \( y = k(x) \), where \( x \in X, y \in Y \), \( x \) represents a given condition, and \( y \) represents the correct selection. Let us use index \( g \) to indicate a particular selection of \( x \) for a given \( y \) (goal). The construction can be expressed now as \( x_g = k^{-1}(y_g) \). Selection of a single unique condition \( x_g \) gives us all necessary information to obtain the output \( y_g \): \( I(x_g) = -\log p(x_g) \), where \( p(x_g) \) is the probability of picking a correct condition \( x \), and \( I \) is the information content of a particular \( x_g \) (Shannon and Weaver, 1949; Brillouin, 1956).

One of the possible ways to refer to the probability distribution \( p(x) \) is to compare it with the reference distribution of the system with maximum entropy and minimum information (equivalent to the thermodynamical state of equilibrium). The state of maximum information entropy we will refer to as a system in abiotic equilibrium. In such a state the probability of a correct selection of a given condition for a given output is uniformly distributed across all the possible conditions, and therefore all selections are equally probable. By calculating the difference between the actual \( p(x) \) and this uniform abiotic distribution, one can calculate the information content needed to make a correct selection.

Unique selection or subset selection. Not only selection of concrete individual causes will code the effect. Any process of constraining the choices from \( X \) to \( A \subset X \) should be treated as information about the purpose. Thus any process of eliminating any causes which do not lead to the effect or lead to the effect with insufficient probability should be treated as information. The information content in such a case is equal to: \( I_X(g; A) = -\log p_X(A) = -\log \sum_A p_X(x) \). This information is called set information in contrast to individual information which is based on selecting a unique individual.
cause $x$. If the set $A$ can be derived from some set of causes, the process of selecting these causes will be equivalent to $I_x(g; A)$ bits of information (if we assume logarithm with the base 2).

In case of non-unique coding $W$ any cause encodes the purpose to some degree. The measure of this is the conditional probability $W(g|x)$. This is the measure of goal $g$ as a result of cause $x$, which we will refer as aptness of $x$. During the test, object $x$ can be provided with the information that its aptness reached a particular level and a particular amount of purposeful information is encoded within the object $x$. The amount of information and the aptness increase together. Process $\{x_i\}$: $W(x_{i+1}|x_i)$ is called a process of improvement if aptness is not decreasing.

### A.2 Code and encoding

The code is a mapping between conditions and a specific reaction to these conditions. In other words Code describes (the macroscopic) behaviour of all objects in a given system. Any given relation between two sets of objects or even conditional probability which binds two sets of objects together, so that selection in one set causes the selection in another, is called a code. The process of mapping one set into another is called: an encoding. It is based on macroscopic laws (e.g. of physics, or the most fundamental computing machine), and represents a traditional cause-effect relationship.

There is an alternative perspective on coding and encoding. Gecow and Hoffman (1983). Each object contains in its structure the information of how to react with the external stimuli – the stimuli from the environment. The lack of such information represents a state of unstable equilibrium point, e.g. one billiard ball placed on top of another (together they form a single object); the upper ball will move toward a particular direction as soon as it will “know” this direction. On the other hand, in a state of equilibrium, all the possible “choices” have the same maximum probability, so the system has the maximum information content.

### A.3 Objects

Let’s consider an isolated system, i.e. a set of situations, from which, based on some criterion, we can isolate an individual object. If from such a situation we subtract the selected object, we will obtain an environment. The environment represents the external conditions in which the object is placed. Object and environment are fully symmetric concepts. Each of them can be treated as the natural code. The other would be treated then as an argument of the encoding, i.e. the cause or the effect. Note also that the separation between the environment and the object is purely operational and is based on the particular criteria that we have chosen as the basis on which to separate them. The most interesting criteria are those cases in which a change of an object has little or no effect on the environment. For our further discussion we will concentrate on such configurations.
A.4  Reversed code

Lets imagine a predefined effect, which is obtained by a particular set of causes. To find causes which lead to a particular effect is to find a reversed coding to a particular code. The reversed coding mechanism we will call a construction. It is simply an inverse of the original code. In many cases, the reversed code is not given explicitly, but it exists only in an implicit form; thus the basic activity of a construction is to test hypotheses which are encoded by the code into effects. This process does not occur naturally in inanimate matter, it is however abundant in the animate world. When testing hypotheses, the construction must compare the expected effect with the actual outcome. Then it records the result as information in its structure. We will discuss this in more detail when discussing hypercycles.

A.5  Naturalistic interpretation

The world out there does not know the word “matter”. Matter is a primary philosophical concept, our axiom that we introduce to deal (to model) the real world. The concepts of “field”, “particle” are derivatives of this axiom, and space and time are also axioms. Therefore if we go down to the basics, (deviating from the applied science which deals with matter casually), we should always keep in mind that we may need to change the axioms if necessary. The world will not change, only its description.

One of the ways to do so is to introduce information as a primary category, which therefore needs no explanation or proof. The information incorporates both material and ideal (it never exists without the material carrier on one hand, but it is not limited to the carrier’s physical properties). Then fields, particles, and such like become the derivatives of the information concept. If we reflect on this a bit, any interaction is in fact an exchange of information.

If we treat laws of physics as an encoding process which translates causes into effects, then the reversed, decoding process, i.e. fitting causes into an assumed effect, is called the process of accumulating purposeful information. If such a decoding process is spontaneous, long, uniform, and effective, then it is called life. Objects participating in the process of life contain the purposeful information encoded in their structure. As a result, properties of such an object, such as its identity, sustained existence, and replication, have been deduced. These derived properties are compatible with the Darwinian mechanisms of random changeability, reproduction and elimination. Furthermore, based on these derived properties of the process of life, the analysis of the structural and morphological tendencies of an evolving system has been performed. It has been shown that the main characteristics of natural life lie in its spontaneity and self-sustainability. On the other hand, any artificial life should be treated as an element of the natural process of life, and as such, constraints on artificial living systems are inherently part of the external setup, and not of natural life itself. The presented discussions provide here the basis for a complete and self-contained deductive theory of life, together with the definitions of the natural and artificial phenomena of life.
A.6 Theory of life and the notion of purpose

The proposed evolutionary model can be interpreted as a theory of life. We start with a set of axiomatic concepts and a set of assumptions, on which the theory builds up. The result is a coherent theory which can be used to investigate properties and to make predictions about the natural and artificial processes of evolution and life. At the centre, there is a notion of a self-maintaining assembly of hierarchically organised dependencies of computational units, and the purpose to exist, that is (implicitly) present in any existing process or structure. We believe that the proposed theory stands out in the simplicity of the assumptions and the accuracy of the predictions. In this work we derive some of the currently known evolutionary mechanisms from a set of simple assumptions. Even though the original assumptions and concerns lie within the realms of computational systems, and outside of biology, there are clear parallels between some of the emergent behaviour of the EVM system, and biological organisms. The work on the EVM architecture was only loosely inspired by biology, yet the recent advances uncover many deep parallels between the EVM architecture and the current state of art in genetics and evolutionary biology.

Based on the formulated axioms, we have investigated the main tendencies of evolving programs (see Chapter 9), and have shown that they are coherent with the tendencies observed in biology. Our goal is to capture all the main observable patterns and on this basis present them as a unified theory of life.

When discussing processes of life and evolution, one inherently uses the term purpose e.g. the purpose of creatures in many artificial evolutionary systems is to maximise their fitness function. In human-driven activities (e.g. technology and engineering) this term does not require any special treatment and fits naturally into our conceptual framework. It is humans who set the purpose for a given system or activity. If we allow everything which is artificial to have its purpose (designed by humans) and if we assume that the purpose of natural living processes is derived in a direct analogy, we would be required to postulate an external creator/designer (e.g. God) who would encode the purpose into the process of natural life.

There is an alternative route to the understanding of the purpose observed in natural life. It follows the work of Varela and Maturana and their theory of autopoiesis. The purpose of the system, is simply to maintain itself. The sole purpose of everything that exists is to persist. Because only processes that can sustain themselves, that can persist, are actually present in the long run and are amenable for observation and investigation, hence the apparent purpose in everything in nature. Similar conclusions where obtained by Gecow (Gecow, 1983a). In our present work we base our notion of purpose as in autopoiesis. A more complete description of autopoiesis is provided in Chapter 7.
A.7 Complexity

One of the prerequisites of the evolutionary computation models is the notion of the “organism” and “environment”. There exist some experimental work investigating the relation between the complexity of the environment and the complexities of the evolved organisms within that environment. The general consensus, which is being confirmed experimentally (Seth, 1998), states that the complexity of environment influences the complexity of the organisms within this environment.

Taking the above thesis to the extreme, and using the basic notions from the theory of algorithmic complexity, we can state that none of the Evolutionary Computation models can in principle create anything beyond the algorithmic complexity that has been introduced into the process from outside. This is a very crucial aspect, and needs more elaboration. But first, what exactly do we mean by “complexity”?

There are many different complexity measures used for many different purposes (Badii and Politi, 1997; Adami, 2002). In our work we use the complexity measure based on the notion of information (Shannon and Weaver, 1949) and algorithmic information (Papadimitriou, 1994; Li and Vitányi, 1997). However, we do not use these in their pure form. We derive the notion of information in relation to the mathematical construct being described. This is very similar to the notion of physical complexity introduced by Adami (Adami, 2002).

In the EC process, the two externally provided sources of complexity are: (a) the externally designed fitness function, and (b) the randomness introduced within the random mutations and crossover operators. It can be shown that any EC process, such as one driven by a genetic algorithm, cannot create a program that is more complex than the complexity of the fitness function and the random number generator.

This follows simply from the definition of algorithmic complexity. The only way to continuously produce higher and higher levels of complexity within the realms of computer programs is to present the system with growing ”randomness” from outside of the system.

This is one of the corner stones of the EVMA. By providing the system with new, more and more difficult problems, we can achieve a truly-open ended evolutionary process, because the external source of information will guide the adaptation and complexity growth within the system itself.
Appendix B

Biological life and the EVMA

Based on our computational model and experimental research with the EVM architecture, we have presented a novel model of the phenomenon of evolution that operates without any scale or fixed topology on many different levels of organisation. Our model is characterised by a power law distribution and exhibits self-organised critical system behaviour. It is too early to provide a definitive answer to the question of whether hierarchies are an inherent part of the phenomenon of biological life, in general. Our models and experiments suggest that a hierarchical model is more accurate than some of the current biological models (without hierarchies). This fact has far-reaching implications. Making explicit assumptions about the character of life hierarchies can help us to better understand the process of biological life. Apart from that, the basic EVM model will help to facilitate new advances in the field of artificial life, and evolutionary computation. It will assist in the design and implementation of more robust and autonomous evolutionary systems. We hope that as more evidence becomes available, the architecture can keep track with advances in evolutionary biology, and help provide more robust and flexible computing systems.

In this chapter we explicitly state some of the implications of the hierarchical organisation of life, and we put forward some questions which could improve our understanding of the phenomenon of life in general.

B.1 Defining life

There are many attempts to formally define the processes of life, and to specify properties of entities participating in such processes. Most of the models are based on the colloquial notions of life and simply list the properties which fit into the common understanding of the process of “life” (Adami, 1997). Some are biology-centric, and apply only to biological life as we know it from Earth (Muller, 1966; Margulis and Sagan, 1995). Other models are more oriented towards the physical sciences, and they build on the notion of living objects as a special case of dissipative structures (Schrödinger, 1945; Prigogine and Stengers, 1980, 1984). Some of the proposals are very narrow and precise (with strong
biological references) (Smith and Szathmary, 1995); some are broader and more abstract (Emmeche, 1992).

There are also formal and systematic attempts to define life, complexity, organism, boundaries and information content based on discrete and cybernetic models (von Neumann and Burks, 1966; Chaitin, 1979; Gecow, 1975b, 1986; Korzeniewski, 2001).

Scale-free property of the process of life. As diverse as the various models of life may appear, most of the models share certain common characteristics. One such shared feature is the lack of scale. The models can naturally be applied to systems and objects regardless of their respective scales and topology of their chain of influences (interactions). It is our belief that this is indicative of an inherent property of the phenomenon of life itself. Note that many definitions implicitly assume just one, particular scale, sometimes even without explicitly acknowledging it, and then analyse and investigate the phenomenon with respect to this single scale. This scale-free property is quite important, and we believe it is a crucial element of any modern theory of evolution, or life in general. It has substantial implications on the artificial life and evolutionary models in the computational realms.

B.2 Testing for life

Among the many existing formulations and definitions of the process of life, one can hardly find a precise test for life, i.e. an effective and objective procedure which would determine a given object or process as being alive or not. What we find, instead, is a mapping procedure. Usually, there is a particular frame of reference, to which a given process or entity is compared. Based on the obtained results the decision is made if the process or entity meets the assumed criteria for life or not. This is, for example, how medical personnel decide if the patient is still alive: the patient is compared to other alive persons and a set of tests is performed if the characteristics are equivalent.

The biggest difficulty is, however, separating one object from another. Objects participating in the process of life, as well as any other complex dynamical system, are all interlinked; and because of this extremely complex dependency network, they are very sensitive to disturbances. Because of the complex interdependencies, it is highly difficult to investigate properties of the individual participating objects. Even though it is often assumed in mainstream evolutionary computation that the surroundings (environment) is constant, the fact is that the environment is interlinked with the species it hosts, and a change in one of the species may cause a catastrophe-like chain of events that reshape the whole landscape of a given environment. Only recently, more accurate co-evolutionary models have been proposed and investigated. This will definitely lead to better understanding of evolutionary (co-evolutionary) processes and of the phenomenon of life in particular.

Most of biology has been based on passive observations of living organisms in their natural habitats (for example a forest, for monkeys, or cells and DNA strands, for cancers). The complex chain of inter-relationships must be observed and well understood. In the past it has been an unwarranted assumption that the individual’s character is independent of the environment. It is now necessary
to acknowledge the intimacy of the interactions between the individual and the environment. This is most likely to be the case with any truly artificial life systems too. The EVM’s ecological environment is an important step into that direction. Even though local manipulations are allowed and necessary, the overall behaviour of the system is self-governed and there is no global control mechanism imposed. The system unfolds on its own, and no-one is directly (globally) in control.

B.3 Symmetries in the definitions

Most of the definitions of life (or artificial life) postulate the notion of the dynamic object contained within a fixed and stable environment. This is often criticised as being inaccurate. Hence, some definitions go further and postulate the notion of implicit relationships between object and environment (early co-evolutionary models).

In the artificial life field and in dynamical systems modelling, it is a common practice to postulate just objects (or entities), without any artificial concept of an environment. Then, naturally, objects are interacting with other objects, and as a result, we have a large network of interdependencies. It is believed to be the most accurate modelling approach for dynamical systems, and it also makes the model more operational and symmetric. It is our belief not only that the phenomenon of life operates without any particular scale, but that it is like such a dynamical system in its nature. Object and environment may be treated interchangeably as active and passive entities, without any violations of particular laws of evolution (or life). It also means that any byproducts of a process of life in one frame of reference, which would not be able to emerge or exist without it, may be treated as active entities autonomously driving the process of evolution in another frame of reference.

Consider the difference between the idea that it is species dynamics driving particular genes to die out or thrive vs. the competing idea that it is the genes alone that drive species to work for them, multiply them, and make them thrive (Dawkins, 1976) (species-centric or gene-centric interpretation of evolution). Both of the above interpretations are valid, in a sense that they fit the data, and both are often used interchangeably throughout the evolutionary literature. Due to the above symmetries and inter-relationships, it should be always kept in mind that there is a frame of reference, often arbitrary, and it is always possible to find a different, sometimes very counter-intuitive perspective on an investigated phenomenon which will be truly compatible with the assumed definitions and data.

The EVM model in itself does not have any fixed notion of a reference, or environment. But, as discussed earlier in experimental setups, a particular frame of reference can be specified outside the actual evolving entities, and it can influence the general dynamics of the system. But metaphorically speaking, this is more like specifying the laws of physics, than it is to specifying just the fitness functions, as in traditional genetic algorithms.
B.4 Life within life

When looking at life forms around us, whether biological, social, or artificial ones, we often decompose them into simpler, yet still living units. We often refer to these living units as life within life. In biology, cancers and parasitic DNA (Orgel and Crick, 1980) are examples of something that can be called life within life (Korzeniewski, 2001). Sometimes single organs or cells are referred to as being alive. They function as some autonomous or semi-autonomous systems within the larger organism. Sometimes we even turn the picture around and treat the small units as the main actors driving the big body under their own command (Doolittle and Sapienza, 1980; Dawkins, 1976). Both perspectives are symmetric and equally valid results of current evolutionary and artificial life theories. We believe that this relativity is an essential aspect of any modern model of evolutionary processes.

A given observer (e.g. a biologist) perceives a subjective hierarchy, and we even try to draw a line between different life forms. However, as sketched in the virtual machine metaphor, without the perfect knowledge of the underlying physical processes, it is very difficult to interpret or to draw definite and objective boundary lines between different hierarchy levels. And in this case, between different life within life. It seems like the whole complex structural hierarchy has to unfold on its own, and cannot be predicted (or calculated) in advance. If this is the case, we are witnesses of this process unfolding in front of our eyes here on Earth all the time. The question is, what scales it is operating on, and if we are capable of observing it at all, across different scales – or, we can only observe it with respect to a single frame of reference (e.g. due to our own limitations in physical space and time scales).

B.5 Communication

Communication can be simply treated as a mechanism for passing signals from an information source to an information sink (Shannon and Weaver, 1949). In many artificial life models it is assumed that the communication is only on a single fixed hierarchy level, and that it is direct. This is an oversimplistic representation of what we can observe in the natural phenomenon of life. Not only is communication performed on different levels simultaneously, but also it is hardly ever direct.

Consider as a simple example person-to-person vocal communication. We can assume that it operates between entities on the same level of hierarchical organisation. Is it direct though? The intended information signal is encoded first into a sequence of neuro-chemical reactions, which then, through nervous connections, control some muscles in the first person’s body. The muscles then modulate and vibrate the air pressed out of the lungs, translating the message into a voice wave. This wave is “passed” through the environment to the receiving set of similarly complicated mechanisms, which at the end translate it back into some neuro-chemical activity. In each step we deal with an enormous amount of noise. If it feels direct and precise, it is only because we mentally project it as such. In reality it is a very indirect and complicated process, spanning multiple levels of organisation.
and utilising multiple semi-autonomous subsystems.

Modelling such processes in artificial life environments is very difficult, if not impossible. The complexity of such models is very high, and the theoretical aspects are not as yet fully understood. We are however getting a better understanding of the role that the environment, communication and implicit relationships play during the development and evolution of living objects. Communication spanning multiple levels of structural hierarchy might be an important aspect of the process, not fully treated in contemporary theories1.

To better understand communication issues, consider the following “company metaphor”. Companies are treated as living units in social systems. They are often described as “emerging”, “struggling to maintain their functions”, or as “trying to survive the selective pressure of the market environment”, or as “fitting into a market niche”. Sometimes we even directly call them alive, until they go broke and cease to exist. Companies may grow, when there is a positive financial surplus of the company operation, and that is invested into the company growth itself. Companies may do things, similarly to individual people, e.g. they can buy goods or land. Companies can be taken to court, and they can be sentenced, if the company did something that was against the law. Regardless of whether they are truly alive, they can be considered so, as a nice metaphor to picture some of the ideas behind different types of communication between hierarchical levels.

How do we communicate with companies? Can we talk to the company, as we talk to our fellow humans? Not really. We have to follow particular channels of communication, e.g. fax, mail or phone, etc. Our communication is then dealt with by a human on the other end; e.g. in direct communication in the company office, we may directly talk with the receptionist. There are also other, more human-less channels, as well, such as automated phone or electronic services, where the human aspect is minimised or completely absent. In the case of direct vocal communication, the information passing does not finish merely on the neuro-chemical activity of the receptionist, but it is forwarded higher in the hierarchy and it is translated into some structural change in the company itself. That upper-level communication may be, for example, a note on someone’s desk, a physical change of the office or a new staff member.

How do companies communicate with each other? Using the same means as described above. Companies are doing business by delegating tasks to the employees and having the communication going through particular human elements. Again, the communication does not finish there – it is initiated and then treated on the company level.

The main point of this metaphor is, that the communication channels, and the means of communication in general, are inherent part of the living forms themselves. Communication is not something external to life, it is not a bare external utility, which living forms simply exploit. It is an integral part of life, and as such, again, it operates on different scales and carries different information for different

---

1Hence for example assuming that there are higher-order life forms in the universe trying to “talk” to us is somehow ill-conceived. We may never be able to talk to higher level life forms, just as we cannot talk to parasitic DNA. If there exist higher-level life forms, they most likely just passively observe us, or manipulate us without us even noticing it.
levels of structural organisation. Even though we may discover other life forms “out there”, we may never be able to communicate with them the way we communicate with other humans. As Wittgenstein once said (Wittgenstein, 1953), even if we knew the language of lions, we would not understand them. Wittgenstein’s point is clearly that because humans and lions do not share the same interdependencies and scales, they cannot share language or understanding. The language (communication channel) is being shaped by all the outward criteria and dependencies, and as such, it is constrained to a particular scale or level or class of entities only.

B.6 Looking ahead

We have presented some of the implications of the current operational definitions of life. We already suspect that the process of life operates on very different scales and dimensions; thus it is difficult or even impossible to define an objective test for life. The currently available theories only allow us to define operational tests, based on some reference frame to which we can compare the phenomenon in question. We have to compare a tested object to some other objects, over a period of time, to be able to designate it as alive. This means that when looking for or communicating with unknown life forms, we would naturally compare them to ourselves, to the life as we know it: the life that is closely perceived as our own life form. This is one of the major pitfalls of the current research in the area of artificial life, mainly because this consideration seems to be pushed aside and neglected.

The hierarchical organisation of living objects provides them with a multitude of communication means, and it is also the reason that the communication itself is subjected to multiple transformations in-between different levels. It is very rare that the communication between living objects is direct, in the proper meaning of the word. Instead, communication is indirect, and is also subjected to multiple translations and a high level of noise. The central idea of this is an emphasis on the communication level – picking a proper level of communication is the key to having the communication established at all. This aspect is again usually neglected, as researchers often make a set of ad hoc implicit assumptions about the level of communication, and neglect the scales, noise, and information transformations and translations.

The inability to predict the properties and levels of organisation from the underlying lower level laws is important. The structural hierarchy levels must “unfold” themselves; it is not possible to “calculate” them in advance. The evolution of the system cannot be predicted in advance. It might be a good exercise to search for emerging and yet unknown “life forms” on our own planet, within our own physical, biological or social systems. Looking at scales lower than ourselves is easy, and we have quite a bit of success there (e.g. microbiology). However, looking at scales “larger” than us is likely to be difficult and challenging, if not impossible. It requires a complete shift of the paradigm (Bloom, 2001). Are we just mere parts of something bigger? Are we simple reactive agents storing and processing information on behalf of a much bigger structure we are not aware of? Is it possible at all, even in principle, to see outside the box we live in?
Appendix C

The EVM’s universe and the Universe

C.1 Introduction

The notion that all (or in weaker sense, some) natural phenomena can be modelled as a giant computable process, some kind of algorithm, has recently gained scientific recognition, and more research is dedicated to the rigorous explorations of the mapping between natural phenomena and the formalised computational systems. There is some debate and controversy as to how much of the natural can be expressed in the models of the artificial, although due to formalised nature of mathematics and physics itself, it is generally accepted that computation is a viable way to model physical reality. Contemporary developments in computer science and in physics not only do no refute computationalism – they provide more data and evidence in support of the basic theses. In this section we discuss some of the aspects of contemporary computationalist efforts based on the traditional notions of Turning Machine computation. Then we present an extended notion of computation, that goes beyond the traditional Turing limit. We propose a new interactive computation model, called Evolvable Virtual Machines (EVMs). The EVM model uses the notion of many independently, asynchronously executing processes, that communicate between each other and with the outside environment. We present some of the pitfalls of traditional computationalism, and compare it to our new, extended computationalist model, based on the notion of massively concurrent interactive computation (hypercomputation). We argue, that hypercomputationism based on the collection of asynchronously concurrently communicating computational machines is a more compact and more appropriate way of representing natural phenomena (or the universe in general). This thesis is theoretically sound and does not violate any of the current state-of-the-art physical theories. We discuss the details of our computational architecture, and present some of the implications of the hypercomputationism on contemporary physical sciences, life sciences, and computer science.

The traditional theory of algorithmic computation is derived from the notion of a person with a pencil and paper, carrying on a step-by-step set of mechanical operations until the final answer is obtained. This formalisation of computation was proposed by Alan Turing (Turing, 1936–7), and...
refined by others, contemporary to Turing, into equivalent computational models (Gandy, 1988). It is important to remember, however, that contrary to common belief, the term *computation* from the beginning was not restricted exclusively to Turing Machine computation.

We refer to the philosophical notion of all processes being computations, as “computationalism”\(^1\). And it refers to the traditional Turing-machine-equivalent computations, i.e. computation at or below the Turing-machine level (Turing, 1936–7). The idea that our universe is, in its core, equivalent to computation was introduced by Konrad Zuse (Zuse, 1967, 1970b). Zuse initiated the field of digital physics, and his thesis lately gained wider acceptance and was popularized most notably by the work of Edward Fredkin and the digital physics programme (Fredkin, 1992). Digital physics postulates that all natural phenomena are equivalent to Turing-level computations. This we refer to as (traditional) computationalism.

Traditionally the term *computationalism* is overloaded with alternative formulations, because of its cognitive science roots, philosophical implications, associations with digital physics, and considerable work on a specific narrower computational scope (the postulate that all natural phenomena are equivalent to purely Turing-level computation). Recently, super-Turing machines, non-enumerable sets theory, and hypercomputation research are adding new dimensions to the field, and we have discussed this in Chapter 4. Some researchers treat hypercomputation as an extended concept of computation, i.e. computation above the Turing limit. To these and others (including us) the concept of computation is not limited purely to algorithmic computation, but covers all possible formalised processes of information processing (or state discrimination, discussed in the Chapter 4). In Chapter 2 we have presented the details of the computational architecture based on the notion of the Evolvable Virtual Machines (EVMs). Our aim here is to extend the notion of computationalism, based on computation above the Turing limit, and we propose a new notion, *hypercomputationalism*.

We try to use the term *computationalism* with its broader philosophical context, denoting the notion of all naturally occurring phenomena being equivalent to Turing-level computation. We use the term *hypercomputationalism* to denote the notion of all naturally occurring phenomena being simply computations, below or above the Turing limit.

Hypercomputationalism can be treated as an alternative model of traditional computationalism, for which we imagine a physical implementation on traditional Turing Machine-like computers, and with all properties equivalent to a Universal Turing Machine (UTM). The potential hypercomputing capabilities, although mathematically sound, may or may not be physically realisable (Aaronson, 2005). However, we argue that appropriate realisable hypercomputational models must be proposed, and experimental data must be collected to make a progress within the field. Our model is based on the notion of non-computable asynchronous timing of the independent processes. Establishing, whether uncomputable time delays between real physical processes\(^2\) is possible in our universe or not, is

---

\(^1\)Note that in one of its forms *computationalism*, in the context of cognitive science, is often referred exclusively to the thesis that mental processes in a human brain are Turing-computable.

\(^2\)For detailed discussion on physicality of mathematical models of physics see also (Svozil, 2003). Note, that the time
beyond the point of this article⁵. The main objective for our EVMA is to provide a broader, more expressive model of computation that can be used effectively to solve computational problems, and to provide more efficient computing architectures. We also aim at providing fundamental framework for asynchronous timing investigations, and possibly, help when faced with uncomputable problems. The actual hypercomputing capabilities of our computational model rely on the physical properties of our universe itself; and so far it has not been conclusively demonstrated experimentally that uncomputable asynchronous timing can be achieved.

So the broader philosophical and physical implications of our model are still of a secondary nature and may or may not be rendered relevant to physics, life sciences and other empirical research areas. Nevertheless, the EVM model does provide an appropriate framework for further experimental studies on computing architectures, and hypercomputation.

C.2 Hypercomputation

Driven by the mechanical processing of human-computers, Alan Turing proposed a device that can simulate their behaviour. This device, called a Turing machine, consists of a tape, divided into discrete squares, and the read-write head, that can move left and right on the tape, one unit at a time. The tape can be initially annotated by marking some squares black, and some white. The colour of the square directly under is recognized by the head, and the head can change the colour of the square, and move left or right. The initial marking of the tape is referred to as a “program”, and the behaviour of the head, represented by a state-transition mapping, characterises the specific description of the machine itself. Although a given machine is fixed, the overall behaviour of the head and the tape depends on the initial marking of the tape. For a detailed description of the Turing machine, refer to (Turing, 1936–7) or any contemporary textbook on the theory of computation, e.g. (Hopcroft and Ullman, 1979). It has been proved (Gödel, 1934) that many different models of computation are in fact equivalent to this simple computing device, and the whole class of computations equivalent to Turing machine is referred to as Turing-level computation.

It is important to remember that from its inception, the Turing model was not the only model of computation and the only formalised model of human thinking. In 1945 Vannevar Bush published an article, “As We May Think”, in Atlantic Monthly (Bush, 1945). In the article, Bush theorised that people do not think in linear structures. This, interestingly enough, contrasts profoundly with what his contemporary, Alan Turing, assumed for his models of computation. Bush proposed a visionary, at delays and synchronisation between physical processes is complicated due to considerations of the Theory of Relativity, and the notion of multiple observers using independent clocks. Hence, global synchronisation is essentially impossible, and this has again a strong influence on practicality and physicality of any computational theory.

⁵In general, the argument is not settled. Some physicists postulate a continuous universe, which would make hyper-computation plausible. However most argue for a discrete universe, equivalent to a giant cellular computer (’t Hooft, 1999, 2005). Even with a discrete universe, it is not clear if the individual elements are synchronised in computable or uncomputable fashion.
the time, model of a computing machine: Memex. Memex was designed for information retrieval and cross-referencing based on high-resolution microfilms coupled to multiple screen viewers, cameras and electromechanical controls. On the design diagrams it looked like a big desk with a camera recording what users wrote and then linking it to other pieces of information indexed in the machine storage space. Bush described Memex as a "device in which an individual stores his books, records and communications, and which is **mechanised so it may be consulted with exceeding speed and flexibility**. It is an enlarged intimate supplement to his memory." (Emphasis added.)

The important point about the Memex machine is that it represents a belief that the way humans think goes beyond Turing computability models. The Memex description, which was written years before the first digital computers had been successfully built and utilised, is a clear reference to what these days would be called **hypercomputation**. Bush’s beliefs are now shared by contemporary researchers from different fields working in the area of hypercomputing (Bringsjord and Zenzen, 2003). Our asynchronous model of the EVM computing architecture has similarities to the principles of the Memex architecture. The collection of independently computing and asynchronously communicating agents is believed to be a more powerful model of computation, and some believe that it closely mimics the way human cognitive processes work.

Any computation that goes beyond that defined by the Universal Turing Machine (UTM) is called **hypercomputation**. Such computation is also known as super-Turing, non-standard, or non-recursive computation (Siegelmann, 1995)\(^4\). Hypercomputing is a relatively new, multi-disciplinary research area spanning a wide variety of fields: computer science, mathematics, philosophy, physics, biology and others.

Hypercomputation provides a sound and consistent framework within the theory of computation, and is as old as the basic Turing model of computation itself. The first conceptual formalisation of hypercomputing machines was done by Alan Turing, himself (Turing, 1938). His original formalism based on the notion of oracles are equivalent to trial-and-error machines and other forms of hypercomputation (Bringsjord and Zenzen, 2003).

Our EVM model, discussed in the following section, uses the combination of the trial-and-error model with asynchronously communicating virtual machines. This allow us to incorporate both the weak and the strong notion of hypercomputing within our framework.

**Computing the uncomputable**. This may sound like an oxymoron, but this phrase is used surprisingly often Bringsjord and Zenzen (2003) and has three sound interpretations within the context

---

\(^4\)Interactive computing in a Turing computable universe is equivalent to a Turing machine. It is functionally equivalent to the proof that the multi-tape multi-head Turing machine is equivalent to the single tape Turing machine (following the Theorem 2.1 from Papadimitriou (1994)). However, in a non-computable universe, a multi-tape model would potentially exhibit non-computable properties and the Theorem does not hold anymore. Also, the theorem relies on the assumption of perfect global synchronisation, that is not applicable in any physical universe due to the nature of Theory of Relativity and multiple observers with multiple clocks. This is because the asynchronously communicating processes do not necessarily follow computable time differences. Therefore, the synchronisation intervals between asynchronous process may become simply a source of uncomputable input, i.e. an oracle.
of hypercomputation.

1. Starting with Turing-machine computation, the goal is to calculate with arbitrary precision an estimated answer to something that is effectively uncomputable by a UTM. This we call a weak notion of hypercomputation.

2. Provide an accurate answer to a Turing-uncomputable problem by utilising oracles (Turing, 1938) or other means of hypercomputation. This we consider strong hypercomputation.

3. In the context of resource bound computation, computing something larger than allowed by the upper resource bound.

Both, strong and weak notions of hypercomputation are mathematically sound and consistent with contemporary theories of computation. The weak notion does not represent any practical difficulty and can be physically built and empirically demonstrated. For examples, see (a) Chaitin’s work on computing the bounds of the definable but uncomputable number $\Omega$ (Chaitin, 2005), and (b) a theoretical trial-and-error machine solving the halting problem (Bringsjord and Zenzen, 2003).

C.3 Evolvable Virtual Machines

There is an increasing amount of work conducted independently within traditional computationalism, within digital physics, cellular automata, artificial life, and evolutionary computation. Certain properties investigated in those diverse settings are invariant and are shared between different complex systems. Our original desire was to integrate some of the recent advances for those diverse fields onto a single coherent theoretical model, together with an experimental framework which could be used for some practical investigations.

The Evolvable Virtual Machine architecture (EVM) is a model for the autonomous building of complex hierarchically organised software systems. Originally designed as an artificial evolution modelling system, the EVM stems from recent advances in evolutionary biology, and utilises notions such as specialisation, symbiogenesis, exaptation and computational reflection. Proponents of symbiogenesis argue that symbiosis and cooperation are primary sources of biological variation, and that acquisition and accumulation of random mutations alone are not sufficient to develop high levels of complexity (Margulis, 1970, 1981). Other opponents of the traditional gradualism suggest that evolutionary change may happen in different ways, most notably through exaptation (Gould and Vrba, 1982). Reflection and reification can be used to model these complex evolutionary models on a computational platform. Computational reflection and reification, on one hand, provides a compact and

---

5There are arguments suggesting that the strong notion of hypercomputation is physically possible. For example, see the discussion about cognition and mathematical thinking in (Bringsjord and Zenzen, 2003). However, some strong claims are made opposing hypercomputation, see for example the review and discussion in (Aaronson, 2005).
expressive way to deal with complex computations, and on the other, provides ways of expanding
computations on a given level via the meta-levels, and meta-computations.

The EVM architecture allows independent computing elements to engage in symbiotic relationships, specialise in specific tasks, evolve towards new tasks, and be used in different contexts than those for which it was originally designed. In addition to classical computational tasks, the EVM architecture can be treated as a new hypercomputational model that combines features of trial-and-error machines and the asynchronous communicating processes paradigm. The trial-and-error behaviour is achieved through continuous looping of different hypotheses and their re-evaluating until the desired precision of the hypothesis is achieved. The asynchronous communication aspect provides the (potential) ability of strong hypercomputation. If our universe exhibits non-computable properties, (or if it is continuous in space and time), then non-computable time differences and delays are viable. This can be utilised by the collection of asynchronously communicating machines, hence, providing the strong hypercomputation. In case that our universe is computable, EVM still offers weak hypercomputing capabilities and provides a more compact and general notion within the hypercomputationalism programme.

C.4 Computational Universe

The notion of a computational universe emerges from different disciplines and for different reasons. The precursors of the idea can be traced back to antiquity (Svozil, 2003). In computer science the very first formulation of the computational nature of the universe was formulated by German computer scientist, Konrad Zuse (Zuse, 1970b), and he is credited with the first precise formulation of the principles of the computable universe. Zuse postulated a "computational space", in which our universe is performing the computation. The actually physical space was assumed to have an isomorphic relation to the computational space. There is also an alternative view, in which the actual computational space has no direct relation to the physical space-time.

Many scientists state the inherent belief in the strong interpretation of the Church-Turing thesis, and one often sees claims, like this:

Almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication.

(Wolfram, 2002), pp. 5 and 716-717.

Fredkin (Fredkin, 1992) along with the digital physics programme, pursues the idea that all natural phenomena (including quantum physics) are inherently reducible to a Universal Turing Machine (UTM).

It is our belief that, in principle, all naturally occurring phenomena are in fact reducible to computations (or hypercomputations). We mean here computations in a broader sense, exceeding the notion
of Turing machine computability. One of the motivations for this work has been to propose a model capable of expressing rules of arbitrary natural phenomena in terms of hypercomputation.

In the time of Newton, the universe was depicted as a perfectly working mechanical machine, like a giant clock, with simple rules governing all the interactions of mechanical building blocks. This modelling analogy stretches to contemporary ideas about the universe. For computer scientists the universe looks like a giant computer. However, we go one step forward\(^6\) and propose that this is exactly how the science, and the building up of models must proceed. We cannot discover something out of nothing. We always have to base the next theory, or the next formulation of a phenomenon on existing data and existing models, by the process of mixing the information, and combining the information available at any given time instance. The notion of the universe as a giant computer was not feasible in Newton’s times, because all the necessary notions of computation where not available at that time. The same is happening now to a certain extent. Progress in the area of hypercomputation is enabling the notion of the universe as a giant hypercomputer to emerge. This will inevitably lead to further advances and new ideas, and this is exactly what motivates the research on the EVM architecture – the ability to allow the system to expand, adapt and grow beyond what it’s original design was.

The main contribution of this work is the reflected architecture that can be used to adapt, modify, and self-organise itself. It is an architecture based on the idea of massively concurrent and asynchronously communicating units, deterministic, yet unpredictable and uncomputable in a Turing computation sense. The model, that can potentially unify different aspects of science into a single framework, providing the language and representations for distributed computing, hypercomputing, biology, social sciences, and physics alike.

### C.5 The need for reflection

The biggest limitation of the digital physics paradigm is that it specifies the upper limit on complexity of the universe, and on the language (or framework) used to model the universe. The Turing Machine-based models will never be able to compute, or model anything more than what a Turing machine can. Although it may seem to be an attractive idea, it is actually easier (and safer) to assume that the limit is somewhat higher – that it lies on the hypercomputing level. Because of the nature of the physically realisable virtual machines, this assumption does not actually amount to changing any assumptions about the physical universe. It enlarges, however, the language (the framework) in which the universe is modelled. The EVM architecture, realised on the global network system such as the Internet, will behave as a complicated Turing-machine, if the universe is only in fact a Turing computable process. The formalism, however, is capable of accommodating more than Turing computation. It can provide ways and language for expressing meta structures that go beyond the original Turing

\(^6\)To push the analogy, we actually employ the meta-meta...-tower; hence we go a countably infinite number of steps forward.
level. Moreover, if the universe is not inherently digital and Turing-computable, then the EVM model is capable of exploiting the hypercomputation within the physical world. As pointed out earlier, the weak hypercomputation provided by the EVM model naturally addresses some of the computational issues that require special treatment within more traditional Turing-based models (Bringsjord and Zenzen, 2003).

C.6 Physics and new computationalism

Most physicists claim that the universe, by virtue of its atomicity, is fundamentally discrete. The first observation of Planck confirmed that when he observed that light is not continuous, but instead exists in quanta. It has been argued that continuous physical equations are actually useful approximation of the discrete physics. The discrete nature of reality is the first fundamental prerequisite for Turing computationalism.

Traditional theories make a clear distinction between “things” and “processes”. But it is possible to arrive at an alternative, process-only perspective on reality. The main push in this direction has been made through advances in computer science, in physics (process physics), in the life sciences (Maturana and Varela, 1980) and in evolutionary biology. The process-centred view of the reality is a fundamental pre-requisite for computationalism.

There is an important ontological difference in modelling reality by processes. The actual substrate becomes irrelevant. Computationalism by its nature is process-oriented. All entities, machines, and data structures within the computational perspective are inherently processes.

Traditional physicists generally have difficulty in changing their basic mindsets. For example Vic Stenger, a physicist, argues\(^7\):

However, I cannot think of a single working physical scientist who is a relativist. That handful who are even aware of Kuhn’s work (most would not even recognize his name) scoff at the idea that scientific truth is an arbitrary social convention. They all can produce examples that belie the notion. One of my favorite examples is the magnetic moment of the electron, which is both calculated and measured to one part in ten billion with the two results in perfect agreement. To characterize this spectacular achievement as nothing more than social convention is absurd. The magnetic moment of the electron (and thus the electron itself) is as objectively real as any concept that humans can bring to mind, including the chair you are sitting on.

To a computer scientist's perspective this is too simple. Presented with an unknown computational module, a black box computational process, a scientist will investigate the properties of the module, and, by repeatable experiments establish certain invariant properties of a given computational

\(^7\)Private communication, 2005.
processes of the black box. However, the scientist will never assume that his own model of the computational process is equivalent to the actual process hidden inside the black box. The reason is that there are infinitely many equivalent computational processes that can render the same experimental results all the time. There are even more equivalent models, that would render the same results given a finite amount of test cases. It is important to remember also that some computations can change their behaviour after a particular stimulus is presented. Jumping to the conclusion that the model is the reality, is very unlikely in these settings, and this is probably why most computer scientists are also objectivists in a philosophical sense.

Electrons cannot become something real because a physicist says so. What is real is only the measurement. The model that explains why the measuring device behaves in a particular way, is just a model, and will remain a model. It is very likely that we will continue refining our models into more and more detailed models, further and further detached from common sense and our day-to-day experiences, and they will always remain models.

The analogy of the electron and the chair is a good illustration of the origins of the physical bias and ontological assumptions some scientists have. The belief that models are real stems from the history of scientific endeavours, along with social and historical conventions.

One of the good illustrations of this phenomena in physics is quantum mechanics. There are a number of different, sometimes unrelated and counter-intuitive, interpretations of quantum mechanics. The observations, predictions and equations are in each instance the same – no physical laws or empirical data are violated. However, the models, the story, the narrative, and the interpretation of the observed phenomena are quite different.

The most interesting for us, from the EVM computationalism perspective, is the interpretation of quantum behaviour provided by David Bohm (Bohm, 1995). Bohm offered a causal sub-quantum theory, where still-unknown forces act as the agents to produce quantum behaviour. This is similar on some level to the way once invisible atoms produce thermal behaviour. In Bohm’s model no signals are transferred faster than the speed of light, and thus no violation of relativity is implied. Bohm’s theory restores traditional local determinism, and does not require (unexplainable) quantum randomness. He answers ontological questions about the source of quantum phenomena in a way that is compatible with EVM computationalism. In fact, EVM computationalism is originally inspired by the Bohm’s interpretation of sub-quantum physics. Unlike Bohm’s original formulation, the EVM architecture does not postulate a holistic universe (holistic in a sense that potentially everything can be linked directly with anything else). Due to the nature of the computational processes involved, a holistic universe is mathematically possible, although highly unlikely to be physically realisable. The EVM form of computationalism connects certain parts of the universe and provides only limited

---

8This analogy can be pushed further, comparing traditional randomness-based quantum theories to statistical thermodynamics, and Bohm’s theory to particle physics. However, this analogy is not quite right, as we are dealing here with some more fundamental properties of reality, and the simple reductionist approach into sub-sub-atomic particles may not make sense anymore.
account for non-locality.

Although the models of Bohm and de Broglie (Bohm, 1957) have not yet been empirically confirmed, they represent a sound and general model. They maintain the deterministic nature of the universe, and they are compatible with general notions of computationalism (or hypercomputationalism).

C.7 Randomness

The main element of dissatisfaction with the traditional Copenhagen and similar interpretations of quantum physics is the requirement for inherent, fundamental randomness. At first glance, claims that our universe that is said to be fundamentally discrete, finite, and not continuously divisible, is consistent with the requirement of randomness on the quantum level. These claims do not generally cause any eyebrow-raising. But again, from the point of view of the computer scientist, claims like that seem to be highly speculative, because of the fundamental infinity inherently hidden in the requirement for truly random processes. This comes about because it has been shown that to have randomness in a discrete system, one would require an infinitely long (infinitely complex) computational process (Chaitin, 2005). This is in direct contradiction to the claim of the discrete and finite nature of the universe. In a finite and discrete universe true randomness is not possible. Arguments, then, about a fundamental randomness at the quantum level in a discrete and finite universe are inconsistent. To postulate true inherent randomness at the quantum level one would require a continuous universe, or at least, a discrete universe of infinite extent.

C.8 Observations

The way to make empirical experiments and create models of the physical reality (i.e. to conduct science) is to provide some data, and observe the results of the computation (the black box model). This is the simple premise of computationalism. What may not be obvious is that it opens doors for certain behaviour that is non intuitive. Because the computational process can modify itself, it follows that each measurement may literally create a new reality. This idea is not new, and it has been proposed by scientists and engineers alike (Grand, 2000).

Our EVM architecture can take advantage of this property, and this is where the “evolvable” part is actually exploited. The computational processes can adapt and change, depending on the input that is being fed into the machine. Detailed discussion of this and similar properties would be lengthy and beyond the scope of the present article.
C.9 Summary

Many researchers from different fields follow Einstein in the desire for a deterministic universe. Traditional computationalism is one way to progress the deterministic reductionist programme. However, traditional computationalism, based on Turing computation comes up short in explaining some of the existing phenomena, such as the possibility of physical hypercomputation, cognition (Bringsjord and Zenzen, 2003), and others. To address those issues, we propose the new research programme: hypercomputationalism. Our approach goes beyond the notion of Turing-computation and is based on the notion of multiple asynchronously communicating and self-referencing machines. The new model is based on the trial-and-error hypercomputing model, and on multiple communicating machines. If the universe turns out to be non-computable in nature, our model, unlike traditional computationalism, will be able to cope with that by means of uncomputable delays between massively concurrent interacting computational systems.

An instance of the EVM computational model has now been implemented, and we are planning some large scale experiments on the Internet. If the universe is not computable in nature, it might be possible to exhibit hypercomputation behaviour. The main relevance of the new computationalism is within the field of computer science, and its relevance to physics and biology needs to be established and investigated further.
References


Christoph Adami. Introduction to Artificial Life. Springer Verlag, December 1997.


Charles W. Carter. Private informal communication. carter@med.unc.edu, 2005.


304


Wolfgang Kantschik and Wolfgang Banzhaf. Linear-tree gp and its comparison with other, 2001a. URL citeseer.ist.psu.edu/591862.html.


Mariusz Nowostawski, Lucien Epiney, and Martin Purvis. Self-Adaptation and Dynamic Environment Experiments with Evolvable Virtual Machines. In S.Brueckner, G.Di Marzo Serugendo,


